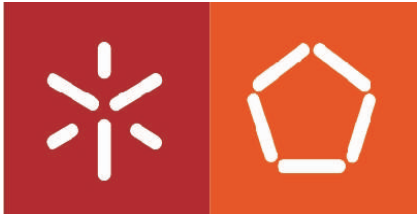
Universidade do Minho
Escola de Engenharia

António Manuel Nestor Ribeiro

Um Processo de Modelação de Sistemas Software com Integração de Especificações Rigorosas

Abril de 2008

Um Processo de Modelação de Sistemas Software com Integração de Especificações Rigorosas
António Manuel Nestor Ribeiro
UMinho | 2008



Universidade do Minho
Escola de Engenharia

António Manuel Nestor Ribeiro

Um Processo de Modelação de Sistemas Software com Integração de Especificações Rigorosas

Tese de Doutoramento
Informática - Fundamentos da Computação

Trabalho efectuado sob a orientação do
Professor Doutor Fernando Mário Junqueira Martins

Abril de 2008

DECLARAÇÃO

António Manuel Nestor Ribeiro

Endereço electrónico: **anr@di.uminho.pt**

Telefone: **253604470**

Título da dissertação/tese:

Um Processo de Modelação de Sistemas Software com Integração de Especificações Rigorosas

Orientador: **Professor Doutor Fernando Mário Junqueira Martins**

Ano de Conclusão: **2008**

Designação do Ramo de Conhecimento do Doutoramento:

Informática - Fundamentos da Computação

Universidade do Minho, Abril de 2008

Assinatura: _____

Agradecimentos

O embarcar numa tarefa desta magnitude corresponde a um esforço que muito dificilmente se pode aguentar sozinho. Apesar de grande parte do esforço ser solitário é reconfortante saber que existem pessoas que estão sempre dispostas a ajudar e apoiar. Nesta fase é conveniente lembrar-mo-nos desses, que em situações de tormenta, ou apenas de esforço continuado, estiveram sempre presentes e constituiram uma ilha de abrigo. Nesses incluem-se aqueles com quem mais directamente se trabalha no dia a dia, mas também aqueles que por serem menos presentes, e mais afastados dos trabalhos de doutoramento, permitem a existência de momentos de bonomia. Esta tese é o resultado de todas essas ajudas, que inequivocamente moldaram, o autor e o conduziram até aqui.

Ao meu orientador, Prof. Mário Martins, pela disponibilidade demonstrada e pelo incentivo constante. As suas sugestões e críticas contribuiram para a melhoria deste trabalho, bem assim como a amizade patente permitiu que me mantivesse sempre empenhado.

Aos meus colegas, de Departamento, pelo ambiente de trabalho e companheirismo demonstrado. Uma lembrança especial para aqueles com quem mais directamente trabalho e que foram sempre incansáveis no apoio prestado. Lembro-me do Costa, Creissac, TóZé, Orlando, Victor, Bacelar e tantos outros que me incentivaram ao longo deste período.

Por fim, nada disto seria possível sem a ajuda da minha família, a qual aguentou sem queixas, os meus períodos de indisponibilidade, de fadiga e de trabalho fora de horas.

À Nocas, pelo apoio de sempre, nos bons e nos maus momentos. Sem ela este trabalho não existiria.

Aos meus pais e irmãos, pela compreensão e apoio nas frequentes ausências, noites perdidas e menor disponibilidade.

Resumo

O desenvolvimento metódico e rigoroso de um sistema *software* é uma tarefa complexa, pelo que deve o engenheiro de software dotar-se de metodologias e técnicas bem fundadas e adequadas a essa complexidade.

Apresenta-se nesta tese uma abordagem à modelação que visa melhorar o projecto e o desenvolvimento de sistemas software complexos, direccionada, essencialmente, aos aspectos relativos à análise e modelação destes sistemas. A proposta dá ênfase, no seu processo de modelação, à captura de requisitos e aos meta-modelos utilizados, dada a reconhecida influência que a informação recolhida na fase de análise tem na qualidade do produto final. Assumindo um processo de modelação baseado em UML, identificaram-se as lacunas referentes à fase de análise, em especial as encontradas no diagrama de Casos de Uso, e propõe-se uma solução para suprir tais lacunas através da adopção de um processo de modelação iterativo.

Este novo processo iterativo baseia-se em: i) construção de uma abordagem unificada com recurso a várias vistas do mesmo modelo, como mecanismo de consolidação semântica da modelação; ii) adição de formalização à descrição dos Casos de Uso, e iii) validação operacional do modelo através de prototipagem.

A construção de uma abordagem multi-vista permite a captura exaustiva de aspectos relativos ao domínio da aplicação, através de um processo de descrição iniciado nos casos de uso. A adição de formalização aos casos de uso através do emprego de uma notação formal, que regista informação sobre os diversos cenários de um caso de uso e as condições necessárias à sua realização, possibilita o enriquecimento do modelo final. A validação operacional permite que a informação recolhida durante a fase de análise possa ser testada através de uma plataforma de prototipagem, fornecendo desta forma informação sobre a qualidade do modelo.

Palavras-chave: análise de sistemas orientada aos objectos, metodologias de desenvolvimento, modelação, UML, casos de uso.

Abstract

The methodical and rigorous development of software systems is a complex task. Therefore the software engineer must use all available resources, namely methodologies and techniques to address and cope with such a complexity.

This thesis puts forward a modelling approach aiming at improving the design and development of complex software systems. This approach is oriented to meet the analysis and modeling needs of such systems. Emphasis is put into requirements gathering by using proper meta-models, due to the well recognised impact that the information collected at this stage has on the quality of the final product. A UML based modelling process is assumed. Analysis related shortcomings are identified, particularly those found in Use Case diagrams. An iterative modelling process is therefore proposed in order to address such shortcomings.

This new iterative process is based on: i) a multi-view unified modeling approach acting as a semantics consolidation mechanism; ii) the formalisation of use case descriptions, and iii) a prototyping based approach to the validation of the model.

The multi-view unified approach enables the comprehensive gathering of features related to the application domain through a description process that begins at use case level. By adding formalisation to use cases the overall model is then enriched. This formalisation is achieved through the use of a formal notation that allows for the recording of information concerning the different scenarios of a use case and also the conditions needed to its execution. The operational validation stage takes the information from the analysis phase and tests it by using a prototyping platform. This provides a more accurate feedback on the quality of the system's final model.

Keywords: object-oriented analysis, development methodologies, modelling, UML, use cases.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A Unified Modeling Language (UML) [Booch 05, Rumbaugh 05, Jacobson 99, Fowler 04] como linguagem de modelação, tem vindo a crescer em importância e utilização, assumindo-se como uma ferramenta importante na análise, modelação e especificação de sistemas de software.

Prova disso é que a UML constitui-se actualmente como a notação mais utilizada para a modelação de sistemas software. Providencia mecanismos que a tornam candidata a cobrir todos os aspectos de modelação de um sistema software, desde a identificação dos actores, entidades e requisitos, até aspectos de concepção arquitectural, abrangendo também a descrição do comportamento das entidades e, inclusive, permitindo modelar características relativas à implementação (*deployment*) do sistema.

A notação assenta principalmente na existência de uma linguagem gráfica, composta por ícones com semântica própria, que permitem efectuar a modelação do sistema. O facto de não se apresentar como uma metodologia, mas antes como uma notação, possibilita também que possa ser utilizada por diferentes metodologias, com a necessária ressalva de que foi concebida baseada em noções próximas do paradigma dos objectos.

A UML apresenta um conjunto de diagramas que permitem conjugar as diversas vistas da modelação de um sistema. Na modelação de um sistema de software existem diagramas com maior capacidade de aplicação do que outros. Nessa modelação não é necessário efectuar a descrição para todos os diagramas, o que origina que existam vistas da UML menos utilizadas, com menor suporte e mesmo com uma semântica mais difusa.

Na modelação de sistemas software complexos sejam estes de larga escala, com necessidades de concorrência, distribuição e interactividade, é possível que uma má

modelação em UML esconda uma parte da funcionalidade e requisitos existentes, inibindo de forma substancial a equipa de projecto, na medida em que esta trabalhará com uma peça de modelação que não corresponde completamente ao sistema que o cliente quer ver concretizado.

Este aspecto é potenciado pelo facto de existirem diferenças substanciais na maturidade e capacidade de expressividade das diferentes vistas que a UML providencia. Parece natural que diagramas estruturais, como os diagramas de classe, estejam bem fundados, pois estão semanticamente muito próximos dos conceitos do paradigma dos objectos e dos construtores existentes nas linguagens de programação. Como se verá em capítulo posterior, quando se abordarem os modelos de processo, a fase de utilização dos diagramas de classe enquadra-se na fase de concepção. Isso significa que os diagramas de classe modelam informação que está já num estágio posterior à fase de análise e assenta em definições bem fundadas, bastante conhecidas e disseminadas.

O suporte da UML para a fase de análise é fornecido pelo modelo de casos de uso, representados graficamente nos diagramas de Casos de Uso ("use case"). No entanto não se pode dizer que a UML seja unanimemente aceite pelos analistas como a escolha melhor fundamentada para a concretização das tarefas de análise, nomeadamente a recolha de requisitos. Não existe uma definição clara de como é que a linguagem pode ser utilizada para a construção do modelo de análise, sendo assim necessário providenciar um suporte mais rigoroso para estas tarefas.

1.1 Modelação Orientada aos Objectos

A inadequação dos métodos não orientados aos objectos tornou-se mais evidente quando foi necessário adequar os modelos às linguagens de programação orientadas aos objectos. Os modelos produzidos pelos métodos estruturados não permitiam uma passagem simplificada para as linguagens de programação por objectos, por não se basearem nos mesmos conceitos, ou princípios fundamentais.

Como forma de obviar esta desadequação surgiram muitos métodos orientados aos objectos, dos quais os mais conhecidos são, o OOAD (Object Oriented Analysis and Design) de Coad e Yourdon [Coad 91], o OMT (Object Modeling Techniques) de Rumbaugh [Rumbaugh 91], o OOSE (Object Oriented Software Engineering) de Jacobson [Jacobson 92] e o método de Booch de Grady Booch [Booch 94]. Cada um destes métodos de análise introduz a sua própria notação e alterações ao processo, sendo que no entanto as diferenças entre eles não se podem considerar como muito substanciais.

Apesar de não serem muito diferentes, o facto de apresentarem uma notação distinta[1] introduzia um grau de entropia que não era desejável. Os clientes e utilizadores, entre os quais as próprias equipas de projecto, viam-se na necessidade de terem de aprender diferentes métodos e notações, exponenciando assim a confusão semântica, já de si crítica em tarefas tão importantes quanto sejam as de modelação.

Com o aparecimento da UML, uma uniformização da notação veio permitir estabelecer uma linguagem padrão (*standard*) para ser utilizada pelas metodologias, nomeadamente as que são orientadas aos objectos. Do ponto de vista da linguagem, a UML é a junção dos esforços de Booch, Rumbaugh e Jacobson, o que se nota no espírito da linguagem, nos diagramas apresentados e no léxico disponível (os ícones e demais simbologia). Dos diagramas da linguagem os mais conhecidos e utilizados são os diagramas de casos de uso, de classes, de actividades, de estados e de sequência. A utilização destes diagramas é prática comum dos engenheiros de software, embora nem sempre os utilizem da melhor forma nem de modo estruturado e organizado.

1.2 Motivação

A Engenharia de Software tem como objectivo agregar uma série de métodos e processos que permitem coordenar o efectivo desenvolvimento de sistemas de software. Esses métodos e processos incutem no desenvolvimento de software rigor e capacidade de raciocinar sobre os problemas e respectivas soluções, funcional e arquitecturalmente, antes de passar à fase de desenvolvimento da solução. Nesse âmbito, são endereçados muitos domínios entre os quais está a recolha de requisitos do utilizador - tanto funcionais como não funcionais - e a captura dessa informação numa representação conhecida e com semântica bem definida.

Essa necessidade é tanto maior quanto maior seja a complexidade do sistema de software que se queira modelar. A recolha de requisitos do utilizador e a sua representação em diagramas de Casos de Uso não pode padecer de problemas típicos de sub-especificação de complexidade. É necessário que nesta fase se identifiquem todas as características do problema, sob pena de as ferramentas depois utilizadas no processo se basearem em informação errada e modelarem estruturalmente conceitos não completamente especificados. Acresce a isto o facto de os diagramas de comportamento serem feitos numa base em que o elemento chave é uma entidade, por decomposição do problema segundo a estratégia normalmente seguida no paradigma dos objectos, o

[1] Embora nalguns casos essas diferenças sejam apenas de pormenor gráfico.

que dificulta a descoberta de sub-especificação a nível da recolha de requisitos.

A necessidade de refinamento do diagrama de casos de uso e das descrições textuais dos casos de uso, quer seja pela incorporação de novos construtores ou pela combinação dos construtores existentes, deriva das lacunas apresentadas pela UML no que respeita à modelação de sistemas complexos e que podem ser brevemente identificadas:

- os diagramas estruturais, diagramas de classe e de objectos, não providenciam um suporte adequado para a modelação, mesmo estrutural, de sistemas com características fortes de interactividade, concorrência e até distribuição. Não é possível pela análise destes diagramas apreender a complexidade do problema;

- os diagramas que permitem a modelação de aspectos dinâmicos, como sejam os diagramas de sequência ou de comunicação (anteriormente de colaboração), não são completamente expressivos no que respeita à descrição de sistemas com interacções mais complexas, devido à pouca flexibilidade da notação a que recorrem. Não é de simples modelação a concretização das actividades que podem ocorrer por uma qualquer ordem, nem é de fácil percepção a modelação de actividades que podem ser repetidas n vezes até se chegar a um determinado estado;

- não é trivial a modelação de actividades em que a interacção de um objecto pode ser afectada pela actividade de um outro objecto;

- não é perceptível em todos os diagramas a distinção entre objectos activos e objectos puramente transformacionais, isto é, que apenas alteram o seu estado através da recepção de mensagens. Não é de fácil modelação, quer a nível estrutural quer a nível dinâmico, a combinação e interacção entre estes dois tipos de entidades;

- por último, e a mais importante para este trabalho, a fase de análise, que está suportada ao nível dos diagramas de casos de uso e respectiva descrição textual, não tem a mesma fundamentação semântica dos restantes diagramas, não sendo de fácil modelação a descrição de sistemas complexos ou apenas com requisitos bastante elaborados. Não é fácil, utilizando apenas os elementos gráficos deste tipo de diagrama e os elementos escritos textuais, perceber pela análise dos modelos criados qual é o grau de complexidade do sistema e quão complexas são as interacções entre as entidades.

Como se verá em maior detalhe em capítulos posteriores, existem trabalhos alternativos que abordam esta problemática. No entanto, este trabalho tem como pressupostos alguns princípios base, que podem ser resumidos em:

1. Dotar o processo de modelação de ferramentas, isto é, diagramas, com maior capacidade de expressividade, de forma a melhorar a documentação obtida na fase de análise. A contribuição é feita ao nível da modelação e do incremento da informação constante nos diagramas de casos de uso, na sua descrição textual e nos diagramas complementares que são utilizados para correctamente se descreverem os requisitos;

2. Foco no uso da linguagem UML como mecanismo linguístico de modelação. Não se propõe no decurso deste trabalho a introdução de novos construtores, na medida em que se considera que uma das vantagens da UML é precisamente o de ter normalizado a linguagem de modelação;

3. Manutenção da semântica do meta-modelo UML, não modificando a semântica de nenhum construtor de modelação, permitindo assim que a comunidade de utilizadores da linguagem não veja as suas expectativas defraudadas no que diz respeito a ter de conhecer variantes da linguagem, e

4. Privilégio da tarefa de análise em detrimento do processo de construção do sistema. O âmbito deste trabalho incide sobre o enriquecimento semântico do processo de análise e não sobre o contributo que o aumento de expressividade esperado possa ter no código final do sistema a conceber, embora pareça evidente que deverá existir uma correlação positiva entre as melhorias introduzidas na fase de análise e a qualidade do produto final.

1.3 Objectivos

O objectivo mais lato da dissertação consiste em providenciar à linguagem de modelação UML - e ao processo de construção de modelos - a capacidade de desenvolver modelos de sistemas de software intrinsecamente complexos que sejam rigorosos e válidos e que permitam o raciocínio e prototipagem ao nível da camada de casos de uso. O trabalho efectuado procurou que essa formulação fosse feita recorrendo às técnicas disponibilizadas pela plataforma (*framework*) da UML de forma a que seja facilmente utilizada nas diversas ferramentas que suportam o processo de modelação.

Numa perspectiva mais fina os objectivos podem ser descritos da seguinte forma:

- criação de um processo de modelação iniciado na descrição dos casos de uso, e que, através de um processo iterativo, constrói o modelo do sistema dando especial ênfase à componente de descrição comportamental;

- incorporação de uma abordagem rigorosa no processo de modelação em UML através da sobre-especificação dos casos de uso recorrendo a expressões escritas numa linguagem de restrições, OCL [Group 06, Warmer 04], que é válida no meta-modelo da UML;

- aumento do nível de expressividade da modelação com casos de uso - complementando a informação diagramática com outra mais rigorosa e bem fundada, que permite, durante a execução desta fase da tarefa de modelação, pensar na orquestração das entidades e no fluxo de informação. Durante a fase de construção do diagrama de casos de uso, à medida que se identificam os actores e as tarefas é possível enriquecer o modelo com expressões que permitem especificar o ciclo de vida das entidades e alterações ao seu estado interno;

- adicionar de expressividade relativa a aspectos complexos de modelação - acrescentando informação aos diagramas e combinando diagramas de comportamento para complementar as descrições efectuadas nos diagramas de casos de uso;

- permitir raciocinar a nível formal sobre os traços de comportamento gerados a partir dos casos de uso - possibilita-se assim um nível de detalhe sobre o modelo geral, ao introduzir na fase de análise elementos necessários mais tarde numa fase de definição de arquitectura. O incremento de informação produzido a este nível, permite também verificar os caminhos possíveis (traços de execução) para as diversas funcionalidades do sistema que está a ser modelado;

- elaboração de uma camada de prototipagem para a animação da execução dos casos de uso - através de uma plataforma de suporte que permita validar operacionalmente as expressões de comportamento dos casos de uso. Possibilita-se desta forma que o cliente possa aquilatar da correcta captura dos requisitos por parte da equipa de projecto.

Com os objectivos apresentados atrás, pretende-se ter como resultado final deste trabalho uma plataforma de raciocínio que permita estabelecer uma forma de aplicação do processo de software mais rigoroso e com maior sustentação dos modelos produzidos. Pretende-se desta forma criar uma plataforma lógica (no sentido de infra-estrutura e de metodologia), que acrescente mais informação, rigor e formalismo, tornando o processo de desenvolvimento de sistemas software mais bem fundado.

1.4 Contributos

As principais contribuições técnico-científicas deste trabalho encontram-se na criação de um nível de raciocínio mais preciso na modelação de sistemas software complexos. Pretende-se apresentar um modelo de processo de modelação que proporcione ao engenheiro de software a capacidade de utilização da UML, recorrendo a construções com semântica precisa e não ambígua. Para tal o processo incorpora um nível de formalismo na descrição da fase de análise, nomeadamente ao nível dos casos de uso, como mecanismo facilitador da recolha de requisitos.

De forma geral, apresenta-se como vantagem o facto de criar documentos e ferramentas que se albergam na linguagem UML definida, não se criando nenhuma rotura semântica pela inclusão de novos componentes, nem pelo facto de se desenvolverem modelos paralelos noutras linguagens ou paradigmas. Recorre-se a linguagens existentes e a mecanismos previstos na UML para especificar os diagramas que são tipicamente mais informais, e conjugar vários diagramas de forma a capturar todos os requisitos do sistema software, que de outra forma não seriam identificados na fase de análise.

Em resumo, neste trabalho apresenta-se um modelo de processo para a fase de análise de sistemas de software complexos. O trabalho foca as questões associadas à análise e modelação de sistemas de software, nomeadamente no que respeita à fase de análise, recolha de requisitos, validação dos requisitos e consequente modelação. As contribuições do trabalho são genericamente endereçadas por três grandes linhas de acção:

- o reforço da utilização dos casos de uso como mecanismo primordial para a captura dos requisitos, dotando-os de maior rigor através do uso de linguagem formal;
- a construção de uma abordagem unificada, com recurso a várias vistas do mesmo sistema, como mecanismo de consolidação semântica da modelação, e
- uma preocupação em operacionalizar o processo através de recurso a prototipagem.

Apresenta-se de seguida a enumeração das principais contribuições para a área de I&D de Engenharia de Software, resultantes deste trabalho:

- Refinamento do modelo de casos de uso por enriquecimento de informação - permite associar à fase de análise (mais desprovida de fundamentação) métodos e construtores que possibilitam que estes diagramas e respectivas descrições sejam

mais expressivas, correctos e com semântica mais bem definida. Possibilita-se assim que estes possam ser utilizados como ferramenta de análise para a descrição de problemas complexos, o que é reconhecidamente uma limitação da utilização actual do modelo de casos de uso;

- Uso de métodos rigorosos para complementar os modelos UML - permite acrescentar metodologicamente informação aos elementos gráficos constantes de um diagrama de casos de uso, de forma a descrever o fluxo de informação entre os diversos casos e os actores envolvidos. A combinação de descrições de casos de uso com a linguagem de restrições OCL descrevendo as pré e pós-condições, bem como as alterações ao estado interno, conjugada com os diagramas de comportamento gerados, permite que se enderece desde cedo a fase de concepção;

- Possibilidade de prova e raciocínio sobre os *traços* gerados a partir de um caso de uso - acrescenta ao modelo a capacidade de raciocinar sobre as alterações de estado das entidades e formalmente provar que o que foi modelado corresponde ao que realmente se quer fazer. Acrescenta um nível de prova que não poderia ser realizado no actual processo de utilização da UML, no qual apenas é possível testar o sistema depois de implementado;

- Camada de validação operacional para prototipagem rápida de descrições de casos de uso - permite verificar, após a tarefa de construção do caso de uso, se o que foi especificado corresponde ao pretendido. Permite também a comparação de informação resultante da especificação usando notação rigorosa no que respeita à preservação dos invariantes de estado e pré e pós-condições.

1.5 Taxonomia e definições importantes

Numa área tão recente e, talvez por isso, em constante mutação, é difícil cristalizar uma designação única para os conceitos.

É no entanto importante no contexto deste trabalho definir alguns termos que serão doravante utilizados com frequência. Este detalhe é importante, até porque um dos aspectos que a tese aborda está relacionado com os problemas gerados pelo facto de, em determinadas circunstâncias, várias pessoas terem uma interpretação diferente sobre os mesmos conceitos.

1.5.1 Sistema

Um *sistema* é um conjunto de componentes cuja interligação faz com que o funcionamento obtido permita atingir um determinado objectivo. Esta definição, por ser muito lata, permite que os componentes de um sistema maior sejam eles também considerados como um sistema. A definição da fronteira do sistema, onde este acaba e começa, o *habitat* onde ele está inserido, só é possível de ser feita a partir do exterior, por um observador externo, sempre em função do que são as acções (ou resultados) esperados.

Por questões de conveniência é por vezes vantajoso abordar o sistema de diferentes "ângulos", potenciando-se a capacidade de nos focarmos em detalhes particulares do mesmo. Daí que seja aconselhável ao analisar um sistema segmentá-lo em *vistas*, de forma a ser mais facilitada a sua compreensão. Nas diversas metodologias que existem para a especificação e desenvolvimento de sistemas é comum encontrarmos fases em que apenas algumas vistas do sistema são relevantes. Importa por vezes prestar mais atenção à componente arquitectural e concentrar numa vista que ilustre essa vertente; outras vezes é necessário detalhar com maior ênfase a componente comportamental, e a melhor forma de o fazer é criar uma vista onde apenas esses aspectos sejam importantes. A vista do sistema, que em determinado momento se utiliza, é aquela que melhor potencia os aspectos que, em determinada fase, mais interessam à equipa de projecto. Um invariante que deve ser garantido implica que as várias vistas de um mesmo sistema são coerentes entre si.

Sempre que neste trabalho se utilizar a designação *sistema*, significa que nos estamos a referir a um *sistema de software*.

1.5.2 Meta-Modelo

É vulgar nas metodologias existentes falar-se no modelo de um determinado sistema. No entanto, para que haja a certeza que a informação representada no modelo é fiável e rigorosa, o modelo deve estar baseado em regras e elementos bem fundados. Para tal é necessário definir o *meta-modelo*, ou o modelo de um modelo, através de um conjunto de elementos funcionais, estruturais e de composição, bem como as regras que permitem associar todos estes elementos.

Como exemplo de ferramentas a utilizar na definição de um meta-modelo podem referir-se os grafos de fluxo de dados, redes de Petri, diagramas de transição de estados (*statecharts*), linguagens formais, entre outras. Como é expectável, o meta-modelo deve ser rigoroso e preciso de forma a não criar ambiguidades na interpretação dos

modelos do sistema. Como veremos ao longo deste trabalho, no caso da UML este esforço de formalização não foi feito numa primeira fase, o que originou ambiguidades nos modelos construídos que têm de ser colmatadas.

Ademais, o meta-modelo deve ser completo, o que determina que a modelação de um sistema será ela também completa, logo descreverá totalmente o sistema (em todas as vistas do mesmo).

1.5.3 Modelo

Um *modelo* é a conceptualização de um sistema tendo em conta o meta-modelo em que se baseou. Um mesmo sistema pode ter vários modelos diferentes na forma, ou seja, na linguagem utilizada, tendo em conta os diversos meta-modelos que se podem utilizar. Por exemplo, num mesmo sistema podemos ter um modelo baseado em símbolos iconográficos e diagramáticos, correspondente a um meta-modelo (utilizando por exemplo a UML) e um outro modelo algébrico de acordo com um outro meta-modelo existente. Esta versatilidade pode ser utilizada quando um determinado meta-modelo permite detalhar algum aspecto do sistema que não é coberta de forma tão eficaz por outros meta-modelos (que podem até ser os mais utilizados na descrição do sistema).

A capacidade de expressividade de um modelo é directamente imputável à capacidade do respectivo meta-modelo. Se este for completo e formal, o modelo sê-lo-á também.

A possibilidade de se obter para um determinado sistema se obter um modelo *não-completo* e *não-rigoroso*, poderá dever-se ao facto de o engenheiro de software estar a utilizar um meta-modelo que também não possui essas características. Este é um problema que se coloca com o uso da UML para a criação de modelos de um sistema.

1.5.4 A Notação

A concretização de um modelo de um sistema faz-se recorrendo à utilização de uma notação e linguagem de acordo com o meta-modelo. A linguagem está estruturada de acordo com uma gramática própria que define um léxico, uma sintaxe, e uma semântica precisa para cada um dos símbolos e respectiva associação. Uma *notação*, ou linguagem, é o conjunto de todas as "frases"válidas que é possível construir. A notação deve permitir expressar todas as características do meta-modelo, em termos de formalismo e completude.

O modelo de um sistema está expresso numa determinada linguagem que permite preservar as características e definição do meta-modelo. É usual utilizar-se indiscriminadamente o termo modelo ou especificação do sistema numa determinada linguagem, visto que para o engenheiro de software, um modelo não é indissociável da linguagem em que está baseado.

Em resumo, pode-se afirmar que a tarefa de modelar um determinado sistema corresponde a obter um ou vários modelo(s) do sistema. A concretização de um modelo numa determinada linguagem dá origem ao que se designa por *especificação*.

1.5.5 Processo, modelo de processo e metodologia

O tema principal abordado neste trabalho incide sobre a forma de desenvolvimento de um sistema de software. Para tal é necessário definir *processo* como sendo a sequência de acções que conduzem à disponibilização final do sistema. Um processo define a maneira de operar e agir com o intuito de obter um resultado final. No âmbito mais restrito da concepção de um sistema de software, um processo é o conjunto de tarefas efectuadas durante o ciclo de vida do sistema. Tendo em conta esta definição, é no entanto necessário elaborar sobre a forma como estas tarefas se organizam, o que determina a sua ocorrência e a sequência pela qual são efectuadas.

Para tal é necessário introduzir a definição de *modelo de processo*. Um modelo de processo organiza e relaciona a forma como as várias actividades de um processo devem ser efectuadas. O modelo de processo determina a sequência das acções, bem assim como determinam as circunstâncias e condições que possibilitam a passagem para a próxima fase (como defendido em [Boehm 88]).

Uma *metodologia* define um conjunto de regras para a prossecução de determinado objectivo. Metodologia deriva do grego *méthodos*, método, e *lógos*, tratado, e estabelece uma determinada abordagem para o desenvolvimento de um sistema. Independentemente do modelo de processo seguido, a metodologia determina o espírito à luz do qual o processo vai ser executado. Ademais, a metodologia pode impor as técnicas e notações que melhor favorecem a abordagem pretendida [Rumbaugh 91, Booch 94]. Na construção de sistemas é possível encontrar metodologias mais orientadas aos objectos ou outras mais orientadas aos dados e ao fluxo que os altera, sendo que o resultado final será um sistema software que sofreu durante o ciclo de vida relativo à análise, concepção, desenvolvimento e implementação, a influência da metodologia adoptada.

1.6 Considerações linguísticas

Num trabalho de índole tecnológica como este, é quase inevitável o uso de expressões ou palavras que não fazem parte da língua portuguesa. O aparecimento de termos noutra língua, não deve ser interpretado como um desrespeito pela língua portuguesa, mas apenas o constatar que o termo na sua língua de origem (normalmente o inglês) tem um significado reconhecido na comunidade científica, o que significa que a sua semântica é menos ambígua. Por outro lado, a língua não é um corpo imóvel e estanque, pelo que não poderíamos ficar restritos ao léxico existente quando não existiam tecnologias de informação. Além do mais, num mundo tão globalizado como o nosso, é por vezes a expressão em língua estrangeira que expressa de forma mais correcta um conceito e não parece ser (pelo menos ao autor) errada a sua utilização na sua língua de origem.

Esta assumpção não levou a que não tivesse havido um esforço de utilização dos termos em português, quando os há e eles têm por si o significado desejado. Noutros casos utiliza-se o termo na língua de origem, havendo o cuidado de na primeira vez que ocorre o sinalizar a itálico, de forma a destacar a sua génese. Um caso particular ocorre com a utilização de *software*, que o autor considera ser parte integrante do léxico português corrente nas tecnologias de informação e ciências da computação, e é um termo aceite pelo Dicionário da Academia de Ciências.

1.7 Estrutura da Tese

A presente dissertação está organizada em sete capítulos que descrevem a área e apresentam uma proposta de processo de modelação que presta especial atenção à fase de análise.

O Capítulo 1 apresenta a área e a motivação que conduziu ao trabalho. Definiu-se o espaço de intervenção e os objectivos propostos e enumeraram-se as contribuições deste trabalho. Apresentaram-se também algumas definições importantes para normalizar o entendimento do discurso.

O Capítulo 2 aborda os modelos de processo mais importantes na modelação de sistemas software. São apresentadas as suas principais características no que concerne às fases que adoptam bem como os resultados de cada uma dessas fases. As metodologias de desenvolvimento estruturada e orientada aos objectos são brevemente descritas. Aborda-se a importância da fase de análise no desenvolvimento de sistemas, procurando demonstrar a seu criticismo no processo. Identificam-se alguns inconvenientes que po-

dem ser introduzidos pelo facto de às tarefas de análise não ser dada a relevância que estas merecem. Dá-se especial destaque à tarefas de recolha de requisitos como sendo de importância nuclear no desenvolvimento de sistemas. Várias questões associadas ao processo de recolha de requisitos são identificadas, e são referidos os diversos tipos de requisitos associados ao desenvolvimento de um sistema.

O Capítulo 3 apresenta as características principais da linguagem de modelação UML e a sua adequação às metodologias orientadas aos objectos. São enunciadas as características mais importantes da linguagem e é justificado o conjunto de ferramentas e diagramas disponibilizadas. São apresentados os diagramas mais relevantes, para este trabalho, da UML, nomeadamente os diagramas de casos de uso, de classe, estado e sequência, e é descrita de forma sumária a linguagem de restrições OCL. São apresentadas também as dificuldades que um engenheiro de software enfrenta para utilizar eficazmente a UML como linguagem de modelação de sistemas complexos e de larga escala. A necessidade de integração de técnicas formais na linguagem é justificada como estratégia para a melhoria da qualidade dos modelos produzidos. Estuda-se a forma de abordagem da UML na modelação de alguns tipos de sistemas aplicacionais. Apresentam-se também trabalhos alternativos que se debruçam sobre a temática desta dissertação.

O Capítulo 4 apresenta o processo de modelação proposto, com o intuito de permitir um acréscimo de importância da fase de análise, dada pela identificação e descrição dos casos de uso e a sua composição no diagrama de casos de uso. A filosofia proposta apresenta uma sequência de passos que permitem capturar a informação necessária para a captura dos requisitos, a descrição dos cenários associados a um caso de uso, a descrição das actividades que decorrem dele e das tarefas associadas. De forma complementar ao processo iterativo e multi-vista que constitui a fase de análise, enuncia-se o processo de incorporação de rigor na descrição de casos de uso, de forma a recolher a informação implícita que concerne às pré e pós-condições. A escrita rigorosa destas restrições conjuntamente com a especificação dos fluxos de execução possíveis, permite a provar que as condições impostas são respeitadas e possibilita a determinação do resultado do caso de uso.

O Capítulo 5 propõe uma arquitectura que possibilita um ambiente de validação operacional dos casos de uso, através da disponibilização de uma plataforma onde os fluxos de execução podem ser validados através de prototipagem. A plataforma oferece um conjunto de mecanismos que permitem a organização dos objectos envolvidos no diálogo e a definição de como é que o diálogo entre eles se efectua.

O Capítulo 6 apresenta a avaliação da proposta através da exemplificação de como

é que o modelo de processo pode ser utilizado. Este capítulo recorre a dois exemplos e para os casos de uso identificados descreve as expressões de comportamento de acordo com o modelo de processo adoptado e demonstra o processo de anotação de expressões rigorosas. Por fim apresentam-se algumas abordagens semelhantes e quando se justifica faz-se uma comparação crítica com o presente trabalho.

O Capítulo 7 apresenta as conclusões, efectua uma reflexão crítica sobre o trabalho realizado e endereça apontadores que permitem antever trabalho futuro no seguimento desta dissertação.

Capítulo 2

Desenvolvimento de sistemas software

2.1 Introdução

Na actualidade, os sistemas aplicacionais são sistemas muito importantes em qualquer contexto, podendo ser mesmo sistemas críticos. A capacidade de integração de sistemas nas actividades mais importantes das organizações torna-os, efectivamente, indispensáveis ao bom funcionamento das entidades que eles suportam. Pode afirmar-se que o funcionamento das organizações depende actualmente do correcto funcionamento dos sistemas de software que sustentam os processos organizativos ou de produção. Este aspecto reforça de forma indelével a necessidade de se providenciarem ferramentas, métodos e metodologias mais rigorosas e fiáveis no processo de construção destes sistemas.

A Engenharia de Software ocupa-se desta vertente e aborda as teorias, métodos e ferramentas que são necessárias para a construção dos sistemas computacionais de software. Apesar de, em teoria, estas "boas práticas" e métodos deverem ser aplicados a todos os sistemas a serem desenvolvidos, é fundamental que o sejam em sistemas complexos, de grande escala, e de complexidade crítica. A engenharia de software não se preocupa apenas com aspectos construtivos e de desenvolvimento, mas dá especial ênfase ao processo de modelação e análise que permite efectuar a especificação do sistema a desenvolver.

A engenharia de software, os seus conceitos e preocupações, foi concebida como forma de combater a propalada crise do software. Esta derivava essencialmente da de-

sadequação dos métodos e processos de desenvolvimento existentes, relativamente aos requisitos cada vez mais complexos dos sistemas que se criavam baseados em plataformas tecnológicas cada vez mais poderosas. Uma das grandes ilações que foi possível retirar foi a de que as técnicas e métodos de desenvolvimento de software que se aplicavam a sistemas "pequenos" não se adequavam a sistemas de maior escala.

Todo este processo de aprendizagem deu origem a novas técnicas, notações e ferramentas que permitem reduzir o esforço necessário para a construção de sistemas complexos e de larga escala. No entanto, apesar da existência de técnicas, metodologias e notações, os problemas não estão resolvidos, na medida em que é necessário verificar a correcta adequação dos processos aos sistemas software que se pretendem desenvolver.

Como em todas as engenharias, a engenharia de software não se restringe apenas ao processo de criação de novos produtos, mas também deve ser responsável pela produção com parâmetros de exigência em termos de qualidade e de gestão do processo (inclusive em termos de custos). Para que este objectivo seja alcançado é necessário que os métodos e processos de construção sejam os adequados e que não se negligenciem fases do processo que são eminentemente mais de análise que de concepção e desenvolvimento.

2.2 O processo de desenvolvimento de sistemas

O processo de desenvolvimentos de sistemas software define o conjunto de actividades e consequentes resultados que dão origem a um produto de software. No processo de desenvolvimento existem quatro actividades fundamentais e comuns a todos os processos existentes. Essas actividades são:

1. *Especificação*: onde se explicita quais são as funcionalidades dos sistemas, respectivas restrições e operações a realizar;

2. *Desenvolvimento*: que corresponde à componente produtiva propriamente dita, desde a concepção arquitectural até à codificação numa linguagem de programação. Erroneamente muitas vezes esta fase é tomada como sendo a única sobre a qual é necessário um maior investimento;

3. *Validação*: onde se verifica se o que realmente foi desenvolvido corresponde ao que foi especificado e funciona como pretendido, e

4. *Evolução*: corresponde às tarefas de manutenção evolutiva, ou correctiva, que o sistema sofre ao longo do tempo.

Em [Pressman 97] divide-se a engenharia de software numa estrutura em camadas. Da base para o topo essas camadas são i) Qualidade, ii) Processo, iii) Métodos e iv) Utilitários. Tal significa que tudo o que se fizer na engenharia de software assenta na premissa base da qualidade, constituindo este o factor principal de comparação e avaliação do produto final. Os processos determinam a forma e a *praxis* da engenharia de software. Os métodos especificam a forma como se deve proceder durante as tarefas de análise, especificação, concepção, desenvolvimento, validação e manutenção. Os utilitários que se encontram no topo da pirâmide, permitem a disponibilização de forma automática, ou semi-automática, de um suporte para o processo e para os métodos.

Para que as equipas de desenvolvimento possam de forma sistémica resolver os problemas com que se deparam, é necessária a adopção de uma metodologia que defina qual é o processo, métodos e meta-modelos a utilizar.

Durante muito tempo o desenvolvimento de sistemas de software era uma tarefa relativamente simples e executada por equipas pequenas e uni-disciplinares. O processo de construção resumia-se a i) escrever código e ii) testar e corrigir erros. É por demais evidente que para sistemas de alguma dimensão e complexidade esta não poderia ser a forma adequada de proceder, visto que:

1. não facilita a organização do código, porque este nasce de forma desconexa e ao ritmo dos erros detectados;

2. não é possível estabelecer uma relação directa entre os requisitos do utilizador com o produto desenvolvido. Isso deve-se ao facto de se negligenciar a análise avançando directamente para a fase de concepção arquitectural, e

3. não é possível ter uma visão clara do esforço necessário para a construção do sistema, visto que tal está intimamente ligado à ocorrência de erros e à sua resolução atempada.

Este processo assenta na premissa base que o programador é o elemento central da equipa e é ele que determina o ritmo de execução do desenvolvimento do sistema. Este problema é o resultado da inexistência de uma abordagem rigorosa e que atribua a importância devida às tarefas de análise e especificação, e é um reflexo da juventude da engenharia de software, por contraste com outras disciplinas mais bem estabelecidas.

Este modelo de codificação e posterior correcção teve de ser abandonado pela incapacidade de se adequar ao aumento da escala dos sistemas a desenvolver, ou mesmo a manter [1]. Uma das vantagens induzida pela crise do software foi a percepção de que faltava método na produção de software, sendo necessário definir e estruturar modelos do processo de desenvolvimento. O modelo de processo de desenvolvimento de sistemas especifica como é que as tarefas de desenvolvimento devem ser estruturadas para uma correcta e atempada obtenção do produto final. Os modelos de processo definem a organização e relação das actividades e técnicas do processo de desenvolvimento.

Importa referir alguns dos modelos de processo mais utilizados, de forma a perceber a importância das tarefas de análise e concepção, arquitectural ou não, na qualidade do produto final.

2.2.1 Modelo em Cascata

O modelo em cascata foi o primeiro modelo de processo desenvolvido para a construção de sistemas software. O modelo em cascata, ou *waterfall model*, oferece a capacidade de desenvolvimento de um sistema de forma sequencial sendo constituído por diversas fases teoricamente independentes. A designação de cascata advém do facto de o controlo do processo terminada uma fase, passar ("cair") para a fase seguinte da sequência, como expresso na Figura 2.1.

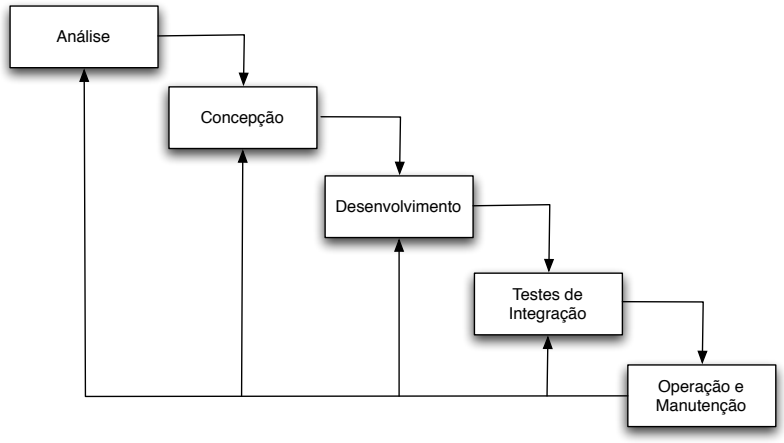


Figura 2.1: O modelo em cascata.

[1] A este reconhecimento e constatação é comum referir-se-lhe como crise do software - *software crisis*.

As principais fases deste modelo, que é muitas vezes denominado como sendo o ciclo de vida do software, são:

1. *Análise* - onde se efectua a análise dos requisitos e a sua definição. Os objectivos, funcionalidades e restrições do sistema são obtidos junto dos potenciais utilizadores do sistema e devem ser descritos de forma que tanto os utilizadores como a equipa de projecto os consigam entender. Esta fase é de importância crucial na medida em que o documento de especificação obtido deve estar construído de forma a não apresentar ambiguidades, visto que a informação recolhida nesta fase é determinante para as fases seguintes, nomeadamente para o desenvolvimento.

2. *Concepção* - definição do sistema software. Nesta fase e em função da especificação existente é conceptualizada uma arquitectura software de suporte que permite o correcto desenvolvimento do sistema. São especialmente importantes os aspectos relacionados com a definição arquitectural do sistema, de forma a facilitar o desenvolvimento e a assegurar maior facilidade e flexibilidade nas tarefas de manutenção evolutiva e correctiva. A definição da arquitectura do sistema determina qual é a estrutura interna do mesmo, baseada na informação do documento de especificação sobre quais são as entidades que o constituem e como se relacionam.

3. *Desenvolvimento*[2] - onde a definição arquitectural efectuada anteriormente é concretizada em termos de linguagens de programação e ambientes de desenvolvimento. Não é suposto que nesta fase os elementos da equipa determinem aspectos relativos à concepção arquitectural do sistema. Nesta fase, que poderia inclusive ser automatizada, a tarefa corresponde a aplicar *boas regras de programação* de forma a construir em código executável os componentes identificados.

   O desenvolvimento inclui também as tarefas de teste unitário. O teste unitário corresponde aos testes que são efectuados no fim do desenvolvimento de forma a determinar se o que foi efectivamente produzido corresponde ao que foi especificado. O teste cinge-se apenas ao componente, ou módulo, em causa e o sucesso do teste unitário não certifica que a inclusão deste componente no sistema seja profícua. O teste unitário apenas permite certificar que o componente está correctamente desenvolvido tendo em conta a especificação fornecida.

[2]Esta fase é por vezes designada por *implementação*, mas como se considera que a implementação é uma tarefa posterior ao desenvolvimento e que corresponde a colocar em funcionamento o que foi anteriormente desenvolvido, neste trabalho será dado uso preferencial a desenvolvimento como definição para as tarefas de construção do sistema.

4. *Testes de integração* - resultante da incorporação de todos os componentes desenvolvidos e respectivo teste de integração funcional. O propósito deste tipo de teste consiste em verificar se a integração de todos os componentes unitários, compondo a arquitectura global concebida, está isenta de erros e se o sistema funciona como especificado.

   Os testes de integração são mais rigorosos e exigentes que os unitários, uma vez que é necessário demonstrar que todos os requisitos de ordem técnica, funcional do sistema são integralmente respeitados.

5. *Operação e Manutenção* - que encerra o ciclo de vida do sistema e que por vezes pode não ser contemplada (depende do sistema em causa). Normalmente o sistema após passar com sucesso pelos diversos tipos de teste é colocado em funcionamento. As tarefas de manutenção e operação envolvem todos os esforços necessários à correcção de erros ou anomalias verificadas durante a exploração do sistema.

   As tarefas de manutenção podem ser também de evolução, pela incorporação ou modificação, de novas funcionalidades. Nessas situações é necessário aplicar o modelo em cascata desde o início, procedendo-se a nova análise.

Na prática todas estas fases estão relacionadas entre si, sobrepondo-se em algumas fases e nenhuma delas pode ser ignorada, ou aligeirada, porque a qualidade do resultado final pode ficar comprometida. O processo de construção de um sistema software não é simples, nem linear, mas baseia-se numa sequência de iterações de actividades. A correcta definição das fronteiras nem sempre é fácil de encontrar nomeadamente no que concerne às fases de análise, concepção e desenvolvimento. Esta separação pode ser melhor, ou pior, conseguida consoante a metodologia que estiver a ser utilizada. A título de exemplo nas metodologias orientadas aos objectos é bastante difícil separar completamente as tarefas de análise e de concepção, visto estarem à luz do paradigma muito relacionadas e ser difusa a fronteira entre elas [Booch 94].

O modelo em cascata, e como expresso na Figura 2.1, inclui iterações constantes (verificar as linhas de realimentação) o que dificulta a identificação de pontos de verificação do estado global do projecto. Esta dificuldade leva muitas vezes à criação de ilhas, nas quais o estado de desenvolvimento é assumido como estável, como mecanismo de aceleração do processo. De forma natural uma das fases mais propensas a esta táctica é a fase de análise, o que por vezes leva a que os problemas sejam esquecidos, ou ignorados, até que seja realmente necessária a sua resolução. Este *modus operandi* conduz, na maioria dos casos, a um esforço maior, uma que vez que a sua detecção é

feita já com o processo muito adiantado.

O maior problema do modelo em cascata advém da sua relativa inflexibilidade no particionamento das actividades, embora numa primeira abordagem seja essa mesma definição clara da fronteira entre as fases que o torna atractivo. É também fonte de problemas o facto de que a verificação e validação só sejam efectuadas após o completo desenvolvimento do sistema, o que em teoria causa impactos fortes no decorrer do projecto. Apesar destes problemas, sentidos particularmente no desenvolvimento de sistemas complexos, o modelo em cascata é um reflexo directo do pragmatismo da engenharia, sendo essa ainda a razão da sua utilização.

2.2.2 Modelo Evolutivo

A noção base do desenvolvimento evolutivo parte do princípio que se desenvolve uma solução inicial, possivelmente incompleta e até genérica, expondo-a aos utilizadores e usando a sua avaliação como mecanismo de refinamento. Este tipo de abordagem pode ser coadjuvada com a incorporação de técnicas de prototipagem de forma a construir versões *intermédias* do sistema.

Este modelo propõe que as actividades de análise, desenvolvimento e validação sejam feitas de forma não sequencial, e até concorrente, com incorporação rápida dos resultados destas actividades na próxima iteração.

Podem identificar-se dois tipos de desenvolvimento evolutivo:

1. *programação exploratória*, em que durante o processo trabalha-se directamente com o cliente de forma a identificar os seus requisitos. O sistema é desenvolvido com as componentes que foram especificadas e é apresentado ao cliente. O modelo evolui pela incorporação de novas funcionalidades propostas pelo cliente.

2. *prototipagem*, em que o principal objectivo é a compreensão dos requisitos do cliente de forma a construir um documento de especificação com a totalidade dos requisitos. O protótipo é utilizado para explorar principalmente os requisitos do cliente que estão mal definidos, ou foram mal especificados.

A abordagem defendida em 1) é utilizada quando é muito difícil, ou mesmo impossível, a concretização de uma especificação detalhada do sistema. Esta abordagem é mais expedita que o modelo em cascata, quando é pretendido obter desenvolvimentos mais rápidos e que satisfaçam uma necessidade muito imediata por parte do cliente.

Contudo, esta abordagem apresenta como problemas:

1. falta de visibilidade do processo - não é possível saber-se, do ponto de vista da gestão do processo, qual é o estado do mesmo, e

2. falta de estruturação do produto - não é possível estruturar convenientemente um sistema que vai sendo construído à medida das necessidades imediatas do cliente.

Com o objectivo de tornear estes problemas (muito difíceis de combater em sistemas complexos), muitas vezes este modelo evolutivo assenta no desenvolvimento de um protótipo. O protótipo é utilizado para compreender e validar as especificações obtidas. A prototipagem inicia-se com o levantamento dos requisitos, dando origem à construção de um protótipo, em que são considerados os aspectos que dizem maioritariamente respeito à interface externa do sistema. O protótipo é avaliado pelo cliente e pode ser gradualmente alterado para incorporar novas necessidades, ou alterações, às funcionalidades existentes. Apesar de o protótipo ser direccionado à captura de requisitos e tendencialmente se esgotar quando a especificação estiver completa, é possível iniciar o desenvolvimento do sistema mesmo não conhecendo totalmente todos os requisitos do sistema. Embora o protótipo não possa ser considerado uma versão final do sistema, por vezes, em sistemas complexos, é possível combinar durante a fase de desenvolvimento o protótipo com código final, de forma a facilitar o desenvolvimento do sistema [Oliveira 95].

A Figura 2.2 apresenta o modelo de processo evolutivo.

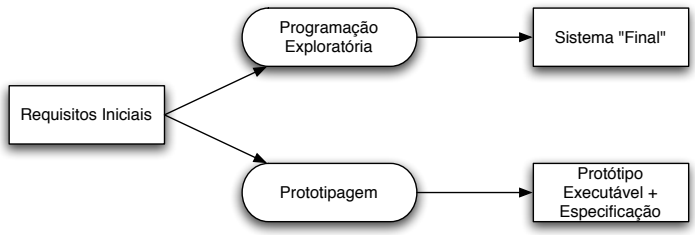


Figura 2.2: Modelo evolutivo.

Modelo Incremental

O modelo incremental é um processo alternativo que combina as vantagens da prototipagem evolutiva com as características do modelo em cascata [Mills 80].

Este modelo de processo consiste na aplicação de sequências lineares de desenvolvimento de forma faseada. Cada sequência produz um incremento no desenvolvimento do produto final. O sistema vai sendo disponibilizado de forma incremental, com o pressuposto que é mais fácil criar uma estrutura mais simples e evolui-la com novos requisitos, do que criar de raiz uma estrutura mais complexa. Tendo em conta o papel da prototipagem no modelo incremental, começa-se por desenvolver os aspectos mais importantes, ou aqueles dos quais se tem conhecimento consolidado, e depois é que se avança para a análise dos restantes.

Obtém-se assim, a capacidade de permitir disponibilizar versões do sistema, estabilizadas e testadas, e iterar repetidamente até o processo estar concluído. Uma desvantagem que este modelo apresenta tem a ver com o facto de a arquitectura do sistema (definida na fase de concepção) ter de ser completamente idealizada antes de os requisitos terem sido todos recolhidos. Este facto pode de alguma forma deturpar o processo pela tendência natural de tentar adaptar os requisitos à arquitectura existente.

2.2.3 Modelo Formal

Um outro modelo de processo que importa considerar é aquele que é baseado em transformações formais. Nele a especificação é transformada, através de uma série de passos que preservam a correcção, num produto final. O produto final satisfaz completamente a especificação, sendo que cada transformação que a especificação sofreu pode ser verificada, para atestar da sua correcção.

Este modelo apresenta a vantagem de as optimizações serem feitas ao nível da especificação, logo mais abstracto, e não ao nível do código. É até perfeitamente possível que o código final seja gerado a partir da especificação. Uma outra mais valia advém do facto de que as provas de correcção de um programa serem difíceis de obter, quando este é complexo e de larga escala. No entanto, num modelo transformacional como o programa é obtido através de uma sequência de pequenas transformações é mais fácil efectuar as provas nas várias transformações.

O modelo de processo formal, devido à especificidade dos conhecimentos que os engenheiros de software devem possuir, é mais útil e porventura mais utilizado, se integrado num ambiente que disponibilize alguma forma de suporte automático para

as actividades que lhe estão associadas.

2.2.4 Modelo em Espiral

O modelo em espiral, ou modelo de Boehm [Boehm 88], é uma abordagem orientada ao risco e dando a este um papel central no processo. O risco é concretizado pela ocorrência de circunstâncias não favoráveis que influenciam de forma negativa o processo de desenvolvimento e a qualidade final do produto obtido.

Este modelo graficamente apresenta a forma de uma espiral e cada ciclo da espiral corresponde a uma fase do processo de software. Cada ciclo é constituído por quatro tarefas: viabilidade, requisitos, concepção e desenvolvimento.

De forma diagramática o custo acumulado do processo está explícito no raio da espiral. A Figura 2.3 apresenta o modelo de forma gráfica.

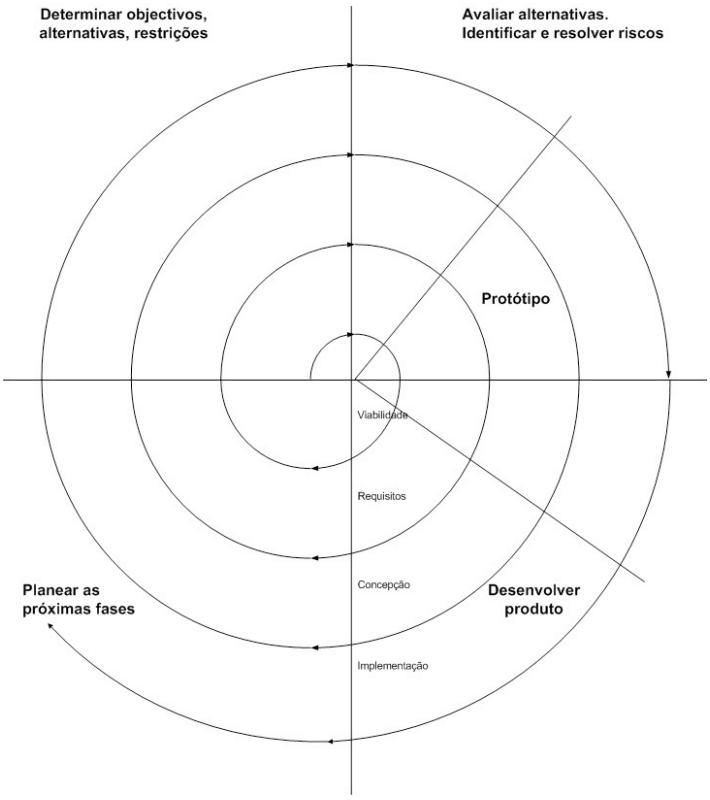


Figura 2.3: Modelo em espiral.

A primeira tarefa do ciclo permite a identificação dos objectivos do sistema que

se pretende produzir. São elencados os condicionantes ao processo e é constituído um plano detalhado do projecto e os riscos deste são identificados. Nesta fase são também levantadas as alternativas possíveis e as restrições associadas a essas alternativas.

A segunda tarefa é dedicada à avaliação do risco e das alternativas identificadas. São efectuadas acções que permitem a redução do risco, como a prototipagem, comparação com semelhantes (*benchmarking*), entre outras.

Na terceira tarefa o sistema é desenvolvido e verificado para a próxima fase. É também nesta altura que se escolhe o modelo de processo mais apropriado para o desenvolvimento[3].

A quarta tarefa pressupõe a revisão dos resultados obtidos e nela planeia-se o próximo ciclo da espiral, se necessário.

O modelo em espiral não impõe que exista apenas um único modelo em cada ciclo da espiral, visto que o modelo incorpora outros modelos de processo. Por exemplo, a prototipagem pode ser utilizada para diminuir o risco associado à captura de requisitos e pode ser complementada com um modelo em cascata para a fase de desenvolvimento.

2.2.5 Rational Unified Process

O *Rational Unified Process* (RUP) é antes de mais a concretização da utilização do *Unified Process* [Jacobson 99] feita pela Rational. A Rational é uma organização que resultou da junção de três dos grandes protagonistas dos métodos de análise, concepção e desenvolvimento orientado aos objectos. Foi efectuado um esforço de súmula das várias notações e fases dos processos defendidos por Grady Booch, James Rumbaugh e Ivar Jacobson, tendo convergido numa abordagem comum, que dá origem a uma ferramenta comercial que usa como linguagem de modelação a UML. A Figura 2.4 dá uma ideia de como se estruturou em termos temporais a criação do RUP.

O Unified Process (UP), como o nome indica é a unificação das várias alternativas que o método de Booch [Booch 94], o OMT [Rumbaugh 91] e o OOSE [Jacobson 92] previa, e a concretização numa abordagem única e com uma notação de modelação também única.

O UP é antes de mais uma plataforma para o processo de desenvolvimento, que deve ser genérico e adaptável. Significa que proporciona um processo pelo qual se pode guiar o desenvolvimento que se fomenta que seja baseado em componentes, de forma a permitir as características desejadas de reutilização e programação genérica. O UP

[3] É por isso que o modelo em espiral é por vezes considerado um meta-modelo.

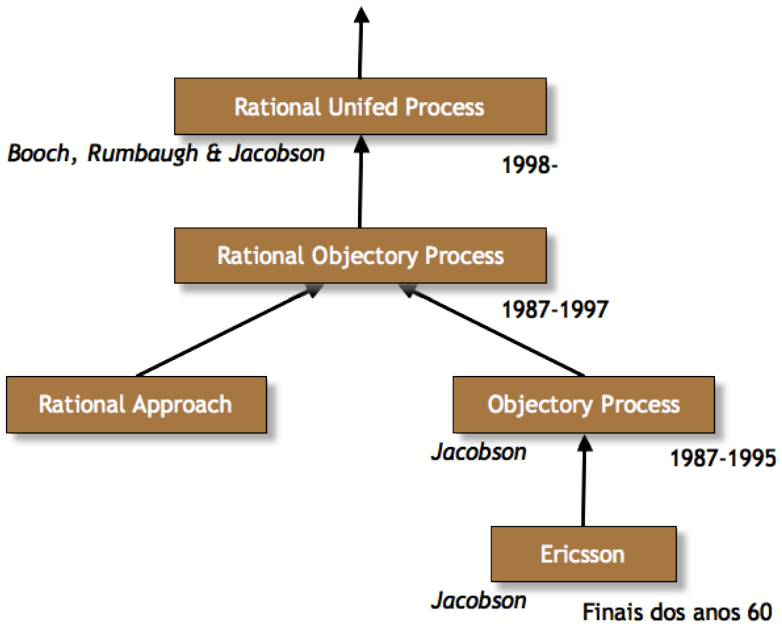


Figura 2.4: História do Rational Unified Process.

utiliza a UML como ferramenta de modelação durante todas as fases do processo de desenvolvimento, isto é, para cada uma das fases que propõe existem mecanismos na linguagem que permitem que esta possa ser utilizada pelo processo.

As ideias chaves do UP podem resumir-se nos seguintes vectores:

1. desenvolvimento guiado por casos de uso;
2. desenvolvimento centrado na arquitectura, e
3. desenvolvimento iterativo e incremental.

O desenvolvimento guiado por casos de uso, induz a captura das interacções entre o sistema e um actor, quer este seja humano ou outro sistema. O modelo de casos de uso é utilizado para:

1. conduzir a concepção do sistema na vertente de captura de requisitos funcionais;
2. guiar a implementação do sistema, no sentido de ter um sistema que satisfaça os requisitos do cliente, e
3. gerir o processo de testes para verificar que os requisitos são satisfeitos.

O desenvolvimento centrado na arquitectura complementa os casos de uso, descrevendo a forma que é necessário construir para correctamente desenvolver esses casos de uso. O modelo arquitectural permite decidir sobre:

1. o estilo arquitectural a implementar, isto é, o tipo de arquitectura que se pretende ter;

2. as componentes do sistema e quais as suas interfaces, e

3. a composição de elementos estruturais e comportamentais existentes.

O desenvolvimento iterativo e incremental possibilita que se possa dividir o desenvolvimento em peças de gestão mais controlada, permitindo diminuir a complexidade do sistema. Em cada iteração é suposto:

1. identificar e especificar os casos de uso relevantes;

2. conceber uma arquitectura que os suporte;

3. desenvolver essa arquitectura, maximizando a utilização de componentes, e

4. testar para aquilatar da satisfação dos casos de uso.

O ciclo de vida do Unified Process é apresentado na Figura 2.5.

O UP divide o desenvolvimento de software em quatro fases distintas, a saber:

1. Início (*Inception*) - onde se identifica o problema, se define o âmbito e natureza do projecto e se efectua o estudo de viabilidade;

2. Elaboração (*Elaboration*) - onde se efectua a análise e concepção lógica do problema, se determinam quais os requisitos e se define, de forma macro, a arquitectura e tecnologia;

3. Construção (*Construction*) - onde se define a concepção física do sistema, sendo esta fase nitidamente iterativa a nível da produção de código envolvendo o desenvolvimento, teste, integração, entre outros, e

4. Transição (*Transition*) - onde se efectuam as tarefas finais de instalação do sistema e se efectuam as tarefas de optimização e estabilização.

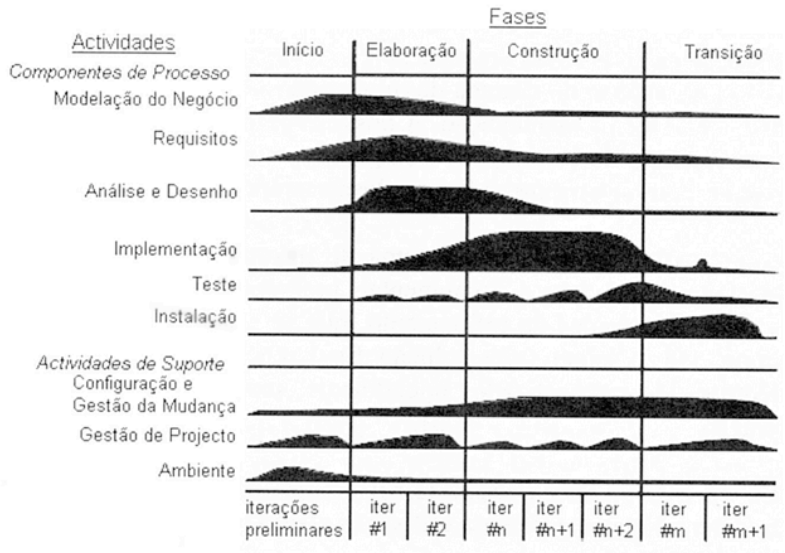


Figura 2.5: Ciclo de vida do Unified Process.

Cada uma destas fases é constituída por um conjunto de iterações, em que algumas das actividades (análise de requisitos, concepção, desenvolvimento, testes, etc.) tem mais peso do que outras, tendo em linha de conta que o objectivo em cada iteração é que seja produzida uma versão final do sistema.

O UP define um modelo de processo de desenvolvimento, sendo que o RUP é uma concretização desse modelo. O RUP fornece:

- ferramentas de gestão do processo de desenvolvimento, segundo a plataforma conceptual definida pelo UP;
- ferramentas para a modelação e desenvolvimento baseadas na UML, e
- uma base de conhecimento.

2.3 Metodologias de desenvolvimento

Após a abordagem de alguns modelos de processo e a apresentação das características que melhor os definem e a forma como neles se enquadram as fases do processo, é necessário apresentar o papel que está reservado às metodologias de desenvolvimento. Como referido numa secção anterior, às metodologias está reservada a definição da abordagem à luz da qual se aplicarão os modelos de processo.

Apresentam-se de seguida as abordagens mais relevantes, prestando-se especial atenção à orientação aos objectos.

2.3.1 Metodologias Estruturadas

As metodologias estruturadas constituíram os primeiros esforços com sustentação para combater a crise do software. Com o aparecimento desta abordagem vários benefícios relativos ao desenvolvimento de sistemas foram sendo introduzidos e cristalizaram-se princípios básicos importantes como: i) a utilização de diagramas como mecanismo preferencial de representação de conhecimento, ii) a decomposição hierárquica como mecanismo de abstracção para tornar mais simples a abordagem ao projecto e iii) a valorização da tarefa de recolha das funcionalidades. A abordagem das metodologias estruturadas baseia-se na filosofia de particionamento do problema em problemas mais pequenos, de forma a serem mais facilmente resolvidos.

Como exemplos das metodologias estruturadas mais conhecidas podem citar-se as de DeMarco [DeMarco 79] e de Yourdon [Yourdon 91], sendo que estas propostas incidem fundamentalmente na modelação do fluxo de dados. Estas abordagens baseiam-se numa separação estrita entre os dados e as funcionalidades (as operações). Esta filosofia, muito evidente, nas linguagens de programação estruturadas, efectua uma distinção muito marcada entre as estruturas de dados e os algoritmos (as operações) que decoram os dados.

A metodologia estruturada tem como principal ferramenta na fase de análise o *diagrama de fluxo de dados* (DFD), que permite a descrição do sistema através de um mecanismo de decomposição funcional. O diagrama de fluxo de dados pressupõe uma decomposição hierárquica do sistema, na medida em que prevê que se possa ter diagramas aninhados noutros de abrangência mais lata. A análise começa com um diagrama mais abstracto, onde o sistema é um único processo com as principais entradas e saídas de dados. Posteriormente este diagrama é detalhado através da descrição de novos diagramas que detalham o sistema a um nível mais fino. Não existe uma definição clara sobre quando (a que nível) se deve parar no processo[4], mas é consensualmente admitido que o processo pára quando as funcionalidades a detalhar sejam de fácil percepção e desenvolvimento. A estrutura de raciocínio, no desenvolvimento de um sistema, é assim conseguida através de uma técnica de refinamento simples, em que se adopta uma estratégia "top-down", ou seja, do mais complexo/abstracto até ao mais simples/concreto.

[4]Alguma literatura defende que com mais de 7 níveis de detalhe já não é aconselhável prosseguir.

As metodologias estruturadas foram bastante bem sucedidas, embora nunca se tenham imposto definitivamente além do nível de documentação, devido a um distanciamento em relação às entidades do problema e um maior enfoque nos dados.

As metodologias estruturadas não tratam a componente dinâmica dos sistemas da forma mais apropriada. Tal deve-se ao facto de estas metodologias estarem mais direccionadas para aplicações transformacionais - orientadas aos dados - nas quais a natural decomposição funcional é suficiente e eficaz [Harel 90].

Tendo em linha de conta a abordagem seguida, as metodologias estruturadas produzem informação na fase de análise sobre: os dados[5], os processos e os eventos. A concepção deve depois fazer a súmula desta informação de forma a produzir um único modelo no qual o desenvolvimento se possa basear. Este facto leva a que se possa afirmar que a separação entre a análise e concepção (e mesmo o desenvolvimento) é acentuada e que a transição entre ambos não é suave, na medida em que não existe refinamento dos modelos entre as fases, mas sim a criação de modelos novos.

2.3.2 Metodologias Orientadas aos Objectos

Assentes na crescente popularidade das linguagens orientadas aos objectos, surgiram as metodologias orientadas aos objectos, nas quais o elemento base de decomposição é a noção de *objecto*.

A decomposição deixa assim de ser eminentemente funcional, como nas metodologias estruturadas, mas foca-se sobretudo nas entidades do sistema e nas suas interacções. O sistema é visto como sendo constituído por uma colecção de objectos que são descritos de acordo com o paradigma nas suas vertentes de estado e comportamento. O facto de se identificarem as entidades, e ao detalhar cada uma delas especificar os dados e o comportamento dos objectos, faz com que as metodologias orientadas aos objectos estejam mais próximos de conceitos relacionados com o domínio da aplicação que as metodologias estruturadas.

Por comparação com as metodologias estruturadas pode-se intuir que a análise e decomposição do sistema através de uma abordagem orientada aos objectos facilita a gestão da complexidade dos sistemas. É mais natural pensar nas entidades que fazem parte do sistema, e para cada uma delas especificá-la individualmente, do que começar por analisar os algoritmos de transformação de dados, como se faz na abordagem estruturada.

[5]Coligida no Dicionário de Dados.

O grão de análise também se reduz devido a que se analisa cada tipo de objectos (entidades) separadamente, o que promove em cada passo a existência de sistemas mais contidos e menos complexos. Como efeito lateral desta constatação também resulta que o grau de reutilização aumenta sobremaneira. É mais simples reutilizar um sistema mais contido e simples (no limite um objecto) que um fluxo de dados (um algoritmo). A granularidade e encapsulamento promovidos desta forma ajudam também a incorporar melhor as alterações às entidades que podem surgir durante o ciclo de vida do sistema.

Também nas metodologias orientadas aos objectos não é fácil encontrar uma que se tenha imposto mais do que as outras. Em 1994 Hutt [Hutt 94] apresentava já uma lista de 21 metodologias orientadas aos objectos. Apesar de todas comungarem dos mesmos princípios base, tipicamente cada um destes métodos de análise e concepção introduz aspectos novos, nomeadamente a nível de notação gráfica, uma vez que quase todas as metodologias apresentam a sua própria notação[6]. O aparecimento da UML [Booch 98, Rumbaugh 98] veio permitir a esta posicionar-se como uma notação única, normalizada pelo OMG (Object Management Group) e que pode ser utilizada pelas diversas metodologias. Note-se que a informação base trocada entre os elementos da equipa de projecto, e mesmo com o cliente, é baseada em informação de especificação, independentemente da metodologia. É pois, uma mais-valia que a notação dos documentos de especificação seja a mesma, qualquer que seja a metodologia, permitindo uma maior familiarização de engenheiros de software e clientes com a documentação produzida.

Por comparação com as metodologias estruturadas, as metodologias orientadas aos objectos são mais facilmente apreendidas, na medida em que estão mais próximas do domínio da aplicação, uma vez que as entidades são representadas por objectos. A análise nestas metodologias está assim mais próxima da concepção e não existe entre estas duas fases uma distância semântica acentuada. O facto de a análise, concepção e as linguagens de programação orientadas aos objectos estarem muito próximas, devido a que se baseiam na noção nuclear de objecto, possibilita uma maior proximidade semântica entre os modelos de todas as fases. Este aspecto potencia o refinamento, e consequente melhoria, do modelo de objectos.

Também se constata que ao existir durante a análise e concepção a utilização de um modelo orientado ao objecto há aspectos que são potenciados, como sejam aqueles que derivam da correcta utilização do paradigma. A saber, a reutilização de componentes, o encapsulamento de estado das entidades e a capacidade de criação de entidades mais complexas por extensão do modelo de objectos existentes.

[6] Apesar de por vezes as diferenças entre notações serem muito ténues.

Em resumo, a utilização do paradigma dos objectos, e mais concretamente das metodologias orientadas aos objectos, apresentam as seguintes mais-valias:

- a divisão de um sistema complexo em unidades mais simples, através da identificação das várias entidades, é mais natural que a decomposição hierárquica típica dos métodos estruturados;
- a simplificação da construção/especificação das entidades ao não terem de se preocupar com os outros objectos, uma vez que a interacção entre os objectos é efectuada por mensagens;
- a potenciação da reutilização de objectos, ou entidades por outros sistemas[7];
- o paradigma dos objectos é uma ferramenta eficaz para lidar com a complexidade dos sistemas;
- a operação e manutenção está facilitada nos sistemas de objectos, devido a que se podem fazer reflectir alterações num componente, não tendo que comprometer o resto do sistema, e
- os sistemas orientados aos objectos são de melhor entendimento, através da consulta da sua especificação, porque lidam directamente com a semântica do problema e facilitam a percepção do domínio da aplicação.

2.3.3 Desenvolvimento baseado em Modelos

Tradicionalmente, a utilização de modelos, reconhecida por todos os intervenientes no processo de construção de sistemas software como sendo valiosa para se representar e pensar o problema, era considerada como sendo desligada do processo de desenvolvimento. A separação entre as tarefas de modelação e o processo de desenvolvimento não beneficia a qualidade do produto final, na medida em que não é obrigatório que a transformação para código siga as indicações do modelo. Com base nesse pressuposto surge a noção de desenvolvimento baseado em modelos, Model Driven Development (MDD), como sendo uma abordagem ao desenvolvimento de software na qual a produção de modelos é a tarefa relevante e a geração de código final é obtida de forma automática através de processos de transformação.

[7]Num determinado domínio de aplicação, por vezes designam-se estes componentes reutilizáveis por entidades de negócio

O desenvolvimento baseado em modelos tem como entidades principais do processo os modelos, as tarefas de modelação e a Model-Driven Architecture (MDA) [Kleppe 03, Raistrick 04] que é uma instanciação (uma *framework*) da MDD.

A abordagem seguida pela MDA separa a funcionalidade do sistema das questões relativas ao desenvolvimento e à tecnologia utilizada. A estratégia passa por manter a modelação ao nível dos requisitos e dos conceitos de negócio e depois através de mecanismos de transformação obter a solução software. O processo que a MDA propõe está expresso na Figura 2.6.

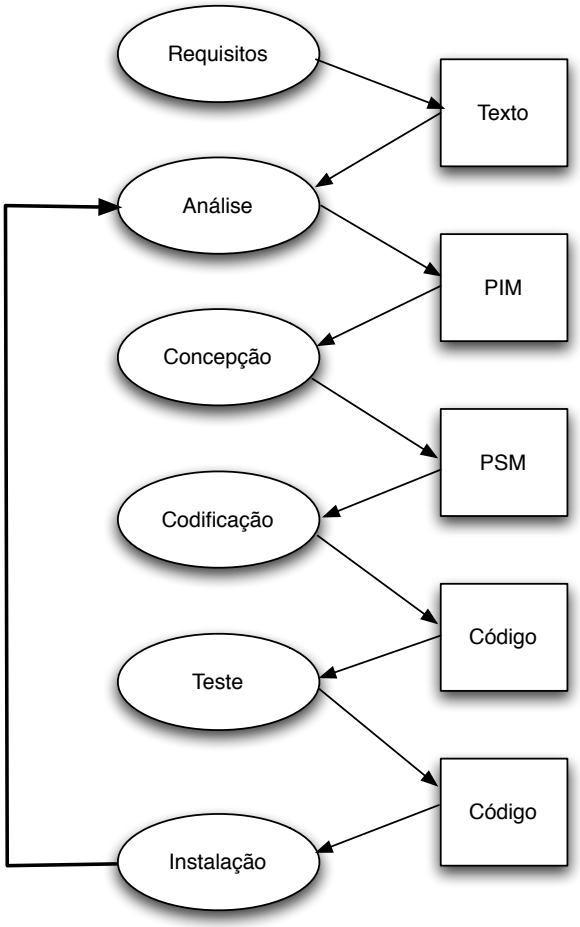


Figura 2.6: Ciclo de vida de desenvolvimento com MDA.

O processo assenta em quatro passos, depois de recolhidos os requisitos:

1. criação de um modelo independente da camada computacional obtido a partir da informação do negócio. Este modelo designa-se por Computation Independent

Model (CIM) e é a concretização do modelo de domínio, onde apenas faz sentido existirem entidades relevantes para o negócio que vai ser suportado pelo sistema software;

2. criação de um modelo da aplicação independente da tecnologia. Nesta fase a UML é utilizada como linguagem de modelação para criar um Platform Independent Model (PIM) que descreve o núcleo dos componentes e serviços oferecidos pela lógica de negócio;

3. desenvolvimento de modelos específicos para determinada tecnologia, acrescentado ao modelo independente da tecnologia os componentes e entidades que seja necessário instanciar. A este novo modelo chama-se Platform Specific Model (PSM) e constitui um refinamento do PIM para uma determinada concretização derivada do suporte tecnológico. A transformação de um PIM num PSM é uma tarefa que pode ser automatizada. Note-se que para um PIM podem existir vários PSM de acordo com as tecnologias alvo, e

4. geração de código, na qual se transforma o PSM em código fonte para determinada plataforma tecnológica.

A realimentação do processo, quando necessária, é feita a nível do modelo de domínio existente no CIM.

2.3.4 Desenvolvimento orientado aos Aspectos

A orientação aos aspectos (Aspect-Oriented) é uma abordagem que tem como objectivo conseguir modularização através da identificação e separação dos interesses que reflectem propriedades fundamentais que pretendemos ter num sistema software. Os interesses são de dois tipos, os que representam um somatório de propriedades e os que representam propriedades básicas intrinsecas. Os primeiros designam-se por *crosscutting concerns* enquanto que os segundos são *non-crosscutting concerns*.

O princípio da separação de interesses (*concerns)* defende que o software deve ser organizado de forma a que cada elemento de um programa apenas se ocupe de uma propriedade e não necessita de ter referências para outros elementos para que seja possível compreender o que representa e faz. A modularidade das linguagens de programação é um princípio de separação de interesses, pelo que os procedimentos, rotinas, objectos, etc., são exemplos desta separação. Note-se que os *concerns* não são detalhes

de programação, mas sim reflexos dos requisitos do sistema e das prioridades e preocupações dos clientes. Exemplos de *concerns* típicos são funcionalidades específicas como segurança, escala, entre outros. Num programa a separação de *concerns* permite estabelecer um fio condutor desde os requisitos até à sua concretização em código, visto não existir mais nenhuma dependência que tenha de ser considerada.

A orientação aos aspectos é um processo de desenvolvimento de software, usualmente utilizado em conjunto com o paradigma dos objectos, que se baseia num conceito de abstracção prioritário que é o "aspecto". Os aspectos encapsulam funcionalidade que atravessa e coabita com outras funcionalidades e incluem a definição de onde devem ser integrados num programa. Apesar dos aspectos terem sido introduzidos como um elemento de programação [Kiczales 97], a noção de *concern* deriva dos requisitos e da importância que estes têm para o sistema que se constrói. Assim, é possível alargar a abrangência dos aspectos a todas as fases do processo de desenvolvimento. Estruturalmente a arquitectura de um sistema orientado aos aspectos é constituída por um núcleo e uma série de extensões, onde o núcleo descreve os principais interesses e as extensões providenciam os interesses ortogonais.

Embora os aspectos tenham surgido como um artefacto ligado à construção de sistemas software são actualmente mais do que um conceito de programação. O desenvolvimento de software orientado aos aspectos (Aspect Oriented Software Development) [Jacobson 05, Filman 04] é um processo de engenharia de software que acompanha o ciclo de vida do desenvolvimento de sistemas software com incorporação de preocupações ortogonais. A Engenharia de Requisitos, que se preocupa com as necessidades e preocupações dos clientes, é também passível de ser integrada numa lógica orientada ao aspecto. A identificação dos requisitos por grupos, os *Viewpoints*, é uma forma de separação dos interesses por diferentes clientes do sistema. Os interesses ortogonais identificam-se com todos os *viewpoints*.

O conceito de aspecto é actualmente bastante abrangente, sendo que os casos de uso podem ser também considerados aspectos, salvaguardadas algumas condicionantes. A definição adequa-se aos casos de uso que são resultado de incorporação por << *extend* >> e numa óptica de engenharia de requisitos a identificação de interesses pode ser feita com o recurso aos casos de uso, em que cada um deles representa um interesse específico e a aplicação define os vários pontos onde esses aspectos podem ser invocados. A identificação dos interesses, tal como a dos casos de uso, pode ser feita através de uma abordagem orientada ao *viewpoint*, isto é, através da segmentação do problema em partes mais pequenas. Nos casos de uso uma estratégia a utilizar será a da decomposição dos requisitos por actor. Em [Schauerhuber 07] encontra-se um

análise de várias abordagens à modelação orientadas aos aspectos.

2.4 A importância da Análise

Pelo atrás exposto percebe-se a grande importância que na construção de sistemas software se dá às fases prévias ao desenvolvimento. É natural que por deformação de formação os engenheiros de software tendam a dar maior importância à concepção arquitectural e posterior passagem a código, do que às fases de análise e concepção. Muitos documentos de especificação apresentam apenas os diagramas de entidades e relacionamentos (E-R) ou os diagramas de classe, sendo considerados pela equipa de projecto como elementos de análise. Como a análise é feita na equipa de projecto por engenheiros de software é expectável que estes pensem no sistema segundo uma visão mais conceptual e arquitectural. No entanto, esta abordagem diminui, ou por vezes elimina mesmo, a fase de análise, cuja diminuição acaba por ser visível no maior custo (dinheiro e/ou tempo) do processo de desenvolvimento.

A Análise pode ser definida como sendo o processo através do qual se efectua a decomposição do todo em partes e se faz o exame de cada uma dessas partes. A análise corresponde ao estudo do problema antes de escolher o que fazer para o resolver [DeMarco 79]. Ao debruçar-se sobre o problema, implicitamente se promove o estudo do domínio da aplicação e constitui-se como a fase em que de forma efectiva se determina *"o que fazer"*. O resultado final da fase de análise é um documento de especificação que descreve de forma i) rigorosa, ii) completa e iii) legível qual a funcionalidade e quais as restrições ao funcionamento. No fim desta fase deve ser possível à equipa de projecto saber exactamente o que o sistema faz, ignorando como é que ele é construído, para exibir o comportamento desejado [Coad 91].

De forma resumida, a análise deve responder, pelo menos, às seguintes necessidades:

1. identificar os requisitos do cliente;
2. determinar as funcionalidades existentes e os actores (humanos ou não) do sistema, e
3. fornecer um documento de especificação que seja utilizado nas fases subsequentes.

A recolha dos requisitos do cliente constitui-se assim como uma tarefa essencial e sem a qual qualquer esforço subsequente não pode produzir os efeitos desejados. A correcta definição dos requisitos do sistema é fundamental na qualidade global do

sistema a ser desenvolvido, além de influenciar, e condicionar, as tarefas de concepção e desenvolvimento.

A tarefa de recolha de requisitos, nunca se afigura como fácil. É necessário ao analista capturar os diversos tipos de requisitos a que o sistema deve responder, ao mesmo tempo que deve apreender o domínio do problema. O esforço deve centrar-se na rigorosa aquisição de conhecimento do sistema, em particular, nos seus aspectos mais importantes. O analista tem como missão a produção de documentação que permita representar o que o sistema faz, não procurando detalhar como é que o faz. O levantamento de requisitos é uma tarefa crucial do desenvolvimento de sistemas, sendo importante investir na melhoria das técnicas associadas [Martins 98]. Sendo a tarefa de importância reconhecida, importa detalhar como é que é operacionalmente concretizada. Além dos aspectos relativos ao processo, como sejam a leitura de documentação, a condução de entrevistas, entre outros, um aspecto de crucial importância é a forma como é que é feita a comunicação entre os analistas e o cliente. O analista deve utilizar técnicas, ferramentas e métodos, que permitam efectuar uma decomposição da complexidade do sistema e ao mesmo tempo capturar todos os requisitos essenciais.

Apesar de existirem diferenças no processo de análise, consoante a metodologia que esteja a ser utilizada, encontram-se etapas e tarefas que são comuns, como sejam a i) descoberta de entidades, ii) a identificação de assuntos, iii) a definição de atributos das entidades e iv) a definição de operações ou funcionalidades. A descoberta destes itens não obedece completamente a uma ordem pré-definida e estanque, mas pode ter interdependências entre eles, de forma a tornar o processo mais eficiente.

2.4.1 Requisitos

Como foi referido a captura e identificação dos requisitos é uma componente essencial da tarefa de análise. Além do processo de recolha, é necessário elaborar sobre a forma como os requisitos são descritos. Muitos desenvolvimentos de sistemas utilizam a escrita em língua natural como mecanismo de comunicação, tornando a informação neles contida ambígua, pouco rigorosa e porventura incompleta. Justifica-se desta forma a necessidade de encontrar uma forma de descrição dos requisitos, através do recurso a um meta-modelo, que permita a modelação da funcionalidade de um sistema. Esse meta-modelo deve prever um conjunto de artefactos e regras de modo a permitir representar a funcionalidade do sistema. O meta-modelo deve ser i) formal e rigoroso, não apresentando ambiguidades, deve ser ii) completo, porque deve prever todas as formas de descrição que permitam representar totalmente o sistema e deve ser iii) legível, para

que todos os intervenientes tenham a mesma interpretação quando os lerem.

Com a introdução da notação UML, um standard OMG, é possível recorrer a uma notação conhecida e aparentemente consistente, para a descrição dos requisitos do sistema. Apesar desta inegável mais-valia, e como se verá em maior detalhe em capítulo posterior, os diagramas de Caso de Uso (*use case*), não são suficientemente expressivos e falta-lhes formalismo, rigor e completude para poderem posicionar-se como mecanismo *universal* para descrição de requisitos.

Os requisitos são registados num documento a que podemos chamar de *especificação de requisitos* e que é o resultado da análise do domínio da aplicação e da funcionalidade requerida. Uma especificação de requisitos é pois, um contrato entre os analistas e o cliente ou utilizador do sistema. Como referido quando foram abordados os modelos de processo, é por vezes vantajoso trabalhar o modelo mesmo antes do sistema estar desenvolvido. Permite-se desta forma a validação, mais atempada, dos requisitos e dos resultados dos sistemas. Daí a importância da prototipagem como mecanismo rápido de verificação e evolução dos modelos criados. De modo a que seja possível a concretização de um protótipo é vantajoso que a linguagem de especificação possa ser executada ou seja, que a linguagem utilizada para conjuntamente o utilizador descobrir os requisitos, possa ser animada. Existem linguagens formais [Jones 86, Group 00] que podem ser utilizadas com este propósito, mas não são facilmente assimiladas pelos utilizadores, pelo que a sua utilização não está massificada. A não utilização de linguagens de especificação formais obriga à introdução de uma etapa suplementar, que corresponde à passagem da notação utilizada para a captura de requisitos para uma linguagem de especificação formal. Neste processo, apesar de todos os esforços, podem ser introduzidos erros e imprecisões que são posteriormente propagados para as fases subsequentes. A importância da validação dos requisitos recorrendo a uma abordagem operacional é evidente, mas não se deve correr o risco de tratar a elaboração do protótipo como sendo o sistema final. O protótipo apenas tem a função de validação dos requisitos conjuntamente com o cliente ou os utilizadores finais. Mesmo assim, o simples facto de ser necessário que o protótipo seja executável implica que esse pequeno sistema software tenha de respeitar algumas características de completude e estrutura de forma a ser possível a sua execução. Este esforço extra é por vezes bem-vindo, porque obriga a equipa a detalhar mais aspectos e juntamente com o cliente, pode levar à descoberta de informação escondida ou imprecisa.

A utilização de especificações executáveis, no contexto de protótipo, tem outra vantagem evidente: a necessidade de considerar todas as alternativas possíveis para o comportamento do sistema. Os utilizadores são assim compelidos a elaborarem to-

dos os *cenários* possíveis, à luz da funcionalidade do sistema e recorrendo a um raciocínio do tipo *"se acontecer X então efectuar Y"*. A descrição algorítmica[8] que é necessário elaborar, permite identificar todas as possíveis formas de resposta e todos os comportamentos possíveis. Este procedimento representa um progresso em relação ao típico processo de recolha de requisitos em papel. Como se verá ao longo deste trabalho, a recolha de requisitos deverá ser feita numa linguagem gráfica universal, de fácil aprendizagem, que possibilite uma passagem correcta e rigorosa para uma linguagem executável, para validação ainda na fase de análise.

A notação para recolha de requisitos pode abordar diversas vertentes consoante a perspectiva que se queira ter do processo. Um sistema pode ser especificado sendo a análise i) baseada em estados, ii) em actividades, iii) em estrutura, iv) em dados ou v) num conjunto das anteriores. Cada uma destas abordagens especifica o sistema de acordo com uma visão do mesmo e por vezes é necessário combinar várias vistas para se ter uma ideia mais concreta do sistema.

2.4.2 Requisitos funcionais e não funcionais

Até ao momento tem-se abordado maioritariamente os requisitos funcionais do sistema. Num primeiro momento, são estes os primeiros a serem especificados, mas existem num processo de desenvolvimento outros requisitos que não concernem apenas ao comportamento funcional, mas ao enquadramento no ambiente em que deve funcionar, a restrições económicas, a características de funcionamento, entre outros. Alguns trabalhos [Morris 96] dividem a classificação dos requisitos em duas grandes famílias: os requisitos funcionais e as restrições à concepção.

Os requisitos funcionais capturam as funcionalidades do sistema, tal como são vistos do exterior. As restrições à concepção podem ser de diversos tipos, nomeadamente *requisitos não funcionais*, *objectivos de concepção* e *decisões de concepção*.

Objectivos de concepção são por vezes confundidos com os requisitos não funcionais, uma vez que especificam características que o sistema deve possuir e que não são especificáveis numa linguagem formal. São usualmente descritos em linguagem natural e tem como papel orientar a equipa de engenheiros de software no desenvolvimento do sistema. A equipa deve tentar incorporar estes requisitos como sendo não funcionais, através da tentativa de quantificação destes objectivos.

As decisões de concepção são decisões de índole computacional e que constituem

[8]Embora os utilizadores não precisem de ter conhecimentos de sistemas software.

restrições à concepção. São exemplos destas decisões, a escolha de hardware, a escolha de sistema operativo, a escolha de componentes, entre outras.

Os requisitos não funcionais representam as qualidades que um determinado produto software deve ter e avaliam o modo como o produto efectua as suas tarefas. A medida destas propriedades não é feita em função das actividades computacionais do produto, mas sim em função das expectativas que o cliente tem em relação ao modo como as actividades são desempenhadas e à qualidade percebida que daí se retira [Robertson 06]. As qualidades não funcionais de um sistema de software constituem-se como um impacto ortogonal a todos os outros requisitos do sistema. São no entanto determinantes para o sucesso do sistema porque a avaliação que o cliente delas faz não é facilmente transformado em informação factual que se possa medir com algum grau de certeza. Os requisitos não funcionais são qualidades globais do sistema software e afectam de forma substancial a sua concepção e subsequente desenvolvimento [Mylopoulos 99].

A área da Engenharia de Requisitos apresenta várias abordagens em que são analisados tanto os requisitos funcionais como os não funcionais. Em relação a estes últimos merece especial relevo citar a *framework* de Non-Functional Requirements (NFR) [Chung 99], cujo elemento central é a noção de *softgoal*, isto é, um objectivo que não possui uma definição precisa nem tem critérios bem definidos para a avaliação do seu grau de cumprimento. Esta parece ser uma definição apropriada a requisitos deste tipo, uma vez que a avaliação destes requisitos não funcionais tem um resultado distinto consoante quem avalia e as circunstâncias em que o faz. Os objectivos que são representados na *framework* NFR são de três tipos, (i) restrições globais ao sistema, (ii) peças de concepção e implementação para satisfazer as restrições do sistema e (iii) explicações para decisões tomadas. Os requisitos não funcionais são representados sob a forma de um grafo onde ficam expressas as interdependências e sinergias entre eles. A *framework* disponibiliza uma função que permite avaliar o impacto das decisões. O seu cálculo é propagado pelo grafo de forma a que se avalie como é que o sistema satisfaz determinados requisitos não funcionais e possa medir os compromissos que é necessário efectuar entre os vários requisitos.

2.5 Resumo

Neste capítulo pretendeu-se dar uma panorâmica sobre a forma como actualmente pode ser abordado o desenvolvimento de um sistema software. É importante perceber como é que é tradicionalmente abordado o processo de disponibilização de um produto, para

perceber as áreas mais críticas e que mais influenciam a qualidade final dos sistemas.

Abordaram-se alguns modelos de processo muito conhecidos e apresentaram-se duas metodologias de desenvolvimento, a estruturada e a orientada ao objecto, que foram e são utilizadas pelas equipas de projecto. Apresentou-se a metodologia de desenvolvimento orientada aos objectos como estando mais próxima do domínio da aplicação, permitindo de forma mais intuitiva, a decomposição da complexidade de um sistema.

Tendo em linha de conta o objectivo deste trabalho, apresentou-se a importância da fase de análise no desenvolvimento de sistemas, procurando demonstrar a sua criticidade no processo. Identificaram-se também alguns malefícios que podem ser introduzidos pelo facto de às tarefas de análise não ser dada a relevância que estas merecem. Deu-se especial destaque à tarefas de recolha de requisitos como sendo de importância nuclear no desenvolvimento de sistemas.

Várias questões associadas ao processo de recolha de requisitos foram identificadas assim como foram apresentados diversos tipos de requisitos existentes num sistema.

Capítulo 3

Modelação em UML

3.1 Introdução

Este capítulo apresenta a linguagem de modelação UML e apresenta as ferramentas diagramáticas que esta coloca à disposição das metodologias que a utilizam. É apresentada a informação que cada um dos diagramas consegue capturar tendo em vista o sistema que se pretende obter. Detalham-se sumariamente os diagramas que são mais importantes para as fases de análise e concepção de um sistema software, o diagrama de casos de uso, de classe, de actividade, de estados e de sequência. Estes diagramas são os mais relevantes para o Modelo de Processo apresentado neste trabalho.

Como a UML é uma linguagem em contínuo desenvolvimento, apresenta-se brevemente as características mais relevantes da versão 2.0, nomeadamente no que concerne ao aumento de precisão e rigor na definição da linguagem. É apresentada também de forma sintética a linguagem de restrições, OCL, que permite decorar os modelos com expressões baseadas na teoria de conjuntos e em lógica de primeira ordem. A utilização da OCL permite adicionar rigor e precisão aos diagramas UML, tornando-os mais explícitos e informativos.

Aborda-se também um aspecto central deste trabalho, na medida em que se elabora sobre as limitações da UML e dos modelos de processo tradicionais na descrição de sistemas, nomeadamente aqueles mais complexos, de maior dimensão e com comportamento dependente do estado. Do ponto de vista estritamente funcional, as facilidades da linguagem permitem que, para sistemas de complexidade controlada a UML possa ser utilizada no processo de desenvolvimento como mecanismo de abstracção e de suporte às fases de análise e concepção. O mesmo não se passa quando os sistemas têm

requisitos funcionais e não-funcionais não triviais, sendo que nesse caso a validade dos modelos pode passar a ser mais aparente que real.

As lacunas existentes na linguagem têm sido supridas de diversas formas no que respeita à necessidade que existe da sua utilização da linguagem em sistemas complexos e de maior dimensão. Existem diversos trabalhos de investigação nesta área que abordam de forma diferenciada, a falta de rigor e precisão da linguagem, e que propõem abordagens diferenciadas para a resolução do problema. Neste capítulo esses trabalhos alternativos são apresentados, indicando-se as grandes famílias de problemas que têm sido alvo de trabalho da comunidade.

Neste capítulo aborda-se a falta de formalismo dos diagramas de casos de uso e exploram-se formas alternativas que podem ser utilizadas como complemento de descrição diagramática.

3.2 A Modelação

A modelação, como referido anteriormente, é uma componente central das actividades que conduzem à concretização de sistemas software bem fundados. A modelação constitui uma técnica de engenharia com provas dadas, visto que a criação de modelos é uma estratégia que permite antever o produto final e raciocinar sobre ele[1].

Um modelo constitui-se tipicamente como uma simplificação da realidade, isolando a informação que queremos analisar e a principal razão pela qual modelamos tem a ver com o facto de que a construção de modelos ser uma forma de melhor compreendermos o essencial do sistema que se está a desenvolver.

Os motivos para que a criação de modelos seja uma actividade profícua podem ser resumidos nas seguintes vertentes [Booch 98, Booch 05]:

- um modelo ajuda a representar um sistema como ele é, ou como queremos que ele seja;
- os modelos permitem especificar a estrutura e o comportamento de um sistema;
- o modelos providenciam uma representação do sistema que ajuda a equipa de projecto na sua construção, e

[1] A modelação não é uma actividade apenas da engenharia de software, mas é também utilizada em todos os ramos da ciência onde a experimentação real nem sempre é possível e um modelo é a forma mais adequada de raciocínio sobre uma potencial solução.

- a criação de modelos permite documentar o sistema e detalhar as decisões tomadas.

Existe a noção, errada, que a criação de modelos só faz sentido para sistemas de larga escala e complexos. É evidente que quanto mais complexo um sistema for, maiores são as vantagens da modelação, mas em rigor pode afirmar-se que qualquer sistema beneficia do facto de dele se criarem modelos.

A modelação é uma tarefa que permite lidar com a complexidade dos sistemas, na medida em que ao construir um modelo o engenheiro de software foca-se nos aspectos mais relevantes do sistema, desprezando aspectos menos importantes. Um modelo é sempre uma vista do sistema, o que é também uma vantagem ao permitir trabalhar o problema a um nível superior de abstracção.

O acto de modelar obedece a princípios simples, mas importantes, como sejam:

1. a decisão de qual é o modelo escolhido tem uma profunda influência na forma como o problema é resolvido;
2. cada modelo pode ser expresso a diferentes níveis de detalhe;
3. não existe um modelo que seja a resposta correcta para um problema. Um sistema, por mais simples que seja, beneficiará sempre da existência de vários modelos independentes entre si.

Em resumo, pode afirmar-se que as actividades de modelação influenciam de forma decisiva a qualidade do sistema final, e que a própria escolha do processo de modelação e dos modelos utilizados estabelece uma orientação metodológica relevante.

3.3 A UML

A Unified Modeling Language (UML) é uma linguagem normalizada para a construção de modelos de sistemas software. A UML pode ser utilizada para visualmente descrever, especificar, construir e documentar os processos que estão na base do desenvolvimento de um sistema software.

A UML é apenas uma linguagem e não é uma metodologia de desenvolvimento, sendo completamente independente do processo de desenvolvimento. Embora tendo em linha de conta a natureza intrínseca dos seus conceitos e a sua raiz no paradigma dos objectos, faz todo o sentido que a modelação em UML seja utilizada num processo

que seja orientado aos casos de utilização, à definição conceptual da arquitectura e que seja iterativo e incremental.

A UML é essencialmente uma notação gráfica, sendo composta por construtores gráficos que simbolizam conceitos, e cuja associação produz *"frases válidas"* na linguagem definida pelo correspondente meta-modelo.

As notações gráficas, apesar de se pretenderem independentes das metodologias existentes e das linguagens de programação por estas utilizadas, acabam por ser fortemente influenciadas no sentido de se mostrarem válidas para o efeito em causa. É claro que ninguém utilizaria uma notação de modelação que tivesse uma distância semântica muito significativa em relação aos conceitos que as ferramentas, linguagens de programação e ambientes de desenvolvimento, utilizam. Este condicionalismo embora impeça uma generalização das linguagens de modelação, tem como factor relevante e positivo, o facto de permitir expressar conceitos numa linguagem visual e iconográfica que podem ser mapeados directamente nas linguagens de programação.

Implica esta característica que mesmo utilizadores familiarizados, mesmo informalmente, com o paradigma e os conceitos, possam trabalhar conjuntamente com as equipas de projecto, não necessitando para tal de terem conhecimentos das linguagens de programação.

Uma linguagem com estas características e com conceitos que são facilmente mapeados nas linguagens de programação acaba por se tornar uma língua franca e a sua utilização aumenta rapidamente. Foi este natural processo de utilização por parte de uma comunidade alargada que fez com que esta fosse uma norma *de facto* e depois uma norma padrão da OMG (Object Management Group).

Como já se referiu a UML é independente das linguagens de programação e do processo de desenvolvimento utilizado, e possui mecanismo de auto-extensão, que possibilita que construtores que não existam na notação possam nela ser incorporados. Esta capacidade de extensão pode ser concretizada através da utilização de estereótipos e de perfis para um determinado domínio de problemas. Durante o processo de desenvolvimento de um sistema software existem um conjunto de actividades que se levam a cabo e nas quais se produzem resultados que são importantes para o processo. Entre os aspectos importantes no processo de software é de destacar a definição dos requisitos, a arquitectura, a concepção, o desenvolvimento em código final, os testes, os protótipos e as versões do sistema que vão sendo disponibilizadas. Alguns destes aspectos são mais do foro da organização do processo em si enquanto que outros são eminentemente fases do processo de desenvolvimento da solução software. No entanto, todos estes aspectos (e não só estes) são importantes no processo de desenvolvimento de um sistema, quer

seja na vertente de produção propriamente dita, quer seja ainda no controlo, medição e documentação.

Conceptualmente a UML pode ser definida como tendo três elementos fundamentais: i) os construtores básicos[2], ii) as relações e iii) os diagramas. Os construtores básicos são as entidades expressas iconograficamente, as relações descrevem a forma como é que elas se relacionam entre si (as regras de relacionamento) enquanto os diagramas capturam os agrupamentos que são possíveis fazer com os dois conceitos anteriores.

No que concerne aos *construtores*, estes podem ser subdivididos em construtores estruturais, de comportamento, de agrupamento e de anotação.

Entre os construtores estruturais encontram-se as *entidades* da linguagem, aquelas que constituem os elementos chave dos diagramas, como sejam as classes, as interfaces, os casos de uso, os objectos, os comportamentos, entre outros.

Relativamente aos construtores comportamentais, estes representam a componente dinâmica dos modelos, as acções. Como exemplos de construtores comportamentais, podem-se referir as interacções, uma sequência de mensagens e as máquinas de estado, que especificam a sequência de estados pelos quais uma entidade passa.

Os construtores de agrupamento são mecanismos da organização dos modelos em UML e representam associações, lógicas ou físicas, de entidades. Um exemplo destes construtores são os *packages*.

Os construtores de anotação correspondem à parte da UML que recorre a notações numa linguagem, em geral não formal, para melhor especificar algum detalhe do modelo ou até aumentar a semântica. Um exemplo típico destas anotações é dado pela componente visual nota (*note*) que permite que nelas se escreva uma nota textual que ajude a compreender melhor o modelo. Normalmente este grau de informalidade não beneficia a qualidade geral dos modelos.

No que respeita às *relações* existem quatro tipos básicos: i) a dependência, ii) a associação, iii) a generalização e iv) a implementação.

As dependências detalham relações semânticas entre construtores indicando que alterações numa extremidade da relação afectam a outra extremidade.

Uma associação é uma relação estrutural entre dois objectos, podendo ser decorada com informação sobre a cardinalidade das ocorrências dos objectos nas extremidades da mesma. Casos particulares de associações são a agregação e a composição de objectos.

[2]Na descrição da linguagem designados como *coisas* - "things".

A generalização implementa uma relação estrutural de hierarquização entre entidades.

Uma implementação é uma relação semântica entre um conceito e uma sua concretização. Um exemplo é a relação entre uma interface e a classe que a concretiza e implementa os métodos aí especificados.

Um diagrama UML é uma representação gráfica de um conjunto de elementos, organizados que se constitui como uma vista do sistema. A UML apresenta como vistas possíveis para um modelo um conjunto de diagramas que capturam informação sobre o sistema.É possível classificar os diagramas em três grandes famílias:

1. Diagramas estruturais - onde se detalha a arquitectura do sistema, quer lógica quer física (em termos de componentes);
2. Diagramas comportamentais - onde se especificam as características dinâmicas de um sistema, e
3. Diagramas de interacção - que constituem um subconjunto dos comportamentais, focando-se nos aspectos de interacção entre objectos.

Nem todos estes diagramas oferecem o mesmo grau de utilidade para a descrição de sistemas complexos. É possível identificar neste contexto aqueles que julgamos serem os mais relevantes, a saber:

1. Diagramas de casos de uso (*use case diagrams*);
2. Diagramas de classes (*class diagrams*);
3. Diagramas de sequência (*sequence diagrams*);
4. Diagramas de estado (*statechart diagrams*);
5. Diagramas de actividade (*activity diagrams*), e
6. Diagramas de objectos (*object diagrams*).

Este subconjunto reforça os aspectos mais complexos das fases de análise e concepção, sendo apresentada na tabela 3.1 a descrição da motivação associada a cada um destes diagramas.

No entanto existem mais diagramas previstos na notação. Dos treze diagramas existentes, nove são originários da notação, nas suas versões 1.x, e os restantes introduzidos

Diagrama	**Motivação**
De casos de uso	Apresenta a identificação dos casos de utilização, bem como os actores e funcionalidades por estes invocadas. É neste diagrama que é representado o resultado do processo de recolha de requisitos.
De classes	Apresenta as entidades computacionais do sistema e determina a relação entre elas. É o diagrama ideal para expressar a fase de concepção arquitectural.
De sequência	Permite representar a forma como objectos interagem entre si para permitir a concretização de determinada funcionalidade. Exprime o algoritmo da funcionalidade e apresenta a troca de mensagens necessária para tal.
De estado	Especifica o comportamento de objectos de uma entidade, permitindo descrever os estados pelos quais passa um objecto e quais são as interacções que desencadeiam essas mudanças.
De actividade	Especifica o fluxo de controlo e as acções para a prossecução de uma actividade relevante do sistema.
De objectos	Permite representar um conjunto de objectos instância e mostrar um exemplo de interacção entre eles. Representa um momento na interacção de objectos (um *snapshot*).

Tabela 3.1: Diagramas da UML.

na versão 2.0 da UML. A tabela 3.2 [Fowler 04] apresenta os restantes diagramas da UML 2.0, não se repetindo aqueles que já foram apresentados anteriormente na Tabela 3.1.

Diagrama	Descrição
De comunicação	Nova designação para os diagramas de colaboração, em que se explicita a troca de mensagens entre objectos.
De componentes	Modela a forma como os componentes que compõem um sistema são organizados e quais são os seus relacionamentos.
De estruturas composta	Permite detalhar de forma mais completa a informação de estrutura de uma classe ou componente.
De Implementação (Deployment)	Mostra a arquitectura física do sistema, detalhando nós, ambientes, *middleware*, etc.
De Interacção	É uma variante do diagrama de actividades, permitindo que exista hierarquização entre os diagramas. Modela o controlo de fluxo num processo de negócio ou num sistema.
De packages	Ilustra a forma como os elementos dos modelos são organizados em associações lógicas, os *packages*, e as dependências entre esses packages.
Temporal	Descreve a forma como uma entidade muda através do tempo, tipicamente como resposta a estímulos baseados na noção de tempo.

Tabela 3.2: Outros diagramas existentes na UML 2.0.

Os diagramas existentes permitem que se aborde a modelação de um sistema através da sua conjugação, cada um deles fornecendo uma vista do sistema e realçando aspectos relevantes (porventura apenas visíveis nessa vista).

Do ponto de vista do modelo de processo de desenvolvimento é possível representar esse esforço conjunto de modelação de aspectos diferentes. A Figura 3.1 explicita o papel de cada uma das vistas no desenvolvimento de sistemas, e identifica claramente a fase de análise e a identificação dos casos de uso como elemento central de todos os esforços.

3.3.1 O meta-modelo da UML

Como referido no capítulo anterior, uma linguagem é uma ferramenta que possibilita a construção de especificações do sistema. A UML permite a construção de modelos e para que esses modelos sejam válidos é necessário que respeitem o respectivo meta-modelo. A normalização pela OMG da UML como linguagem padrão para a modelação

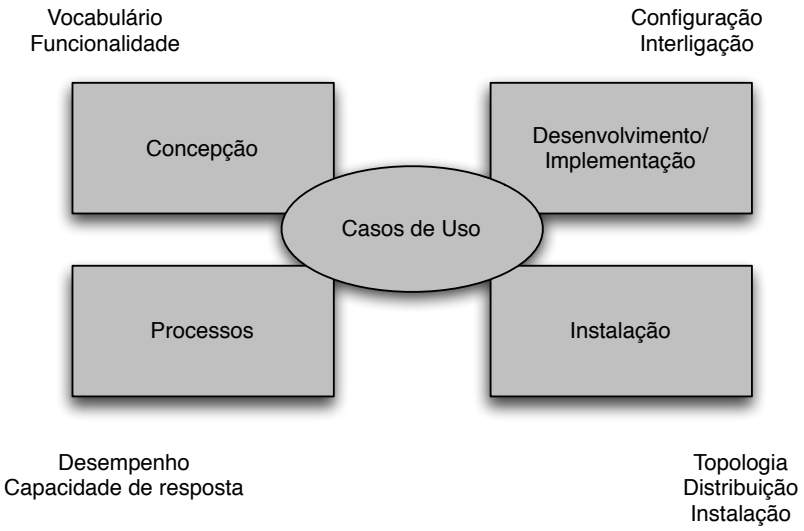


Figura 3.1: O método de desenvolvimento de sistemas na UML.

de sistemas, implica que seja claro o significado de cada modelo construído e que seja interpretado da mesma forma por todos utilizadores da linguagem.

Além da disponibilização da notação, desde o aparecimento desta que existe uma definição do respectivo meta-modelo [Language 97a, Language 97b]. É aliás a única garantia de que seria possível compreender de forma inequívoca os modelos criados.

O meta-modelo do UML está também definido em UML, o que garante, caso seja utilizada uma linguagem formal para o definir, que ele próprio seja rigoroso, sem ambiguidades e com uma semântica clara para os modelos construídos. Apesar desta constatação, não se pode infelizmente dizer que o meta-modelo da UML seja bem fundado e rigoroso e não contenha ambiguidades na definição da semântica dos seus construtores. Isso deve-se ao facto de que o meta-modelo da UML não estar ainda completamente definido formalmente, visto que muitos construtores ainda têm a sua sintaxe e semântica definidas numa linguagem natural.

Apesar de muitos esforços [Evans 98, France 00, vEB98] apenas se pode dizer que a linguagem é semi-formal, uma vez que apesar da sua estrutura da linguagem ser rigorosa a semântica do que está representado é bastante informal [Övergaard 98]. Foi realizado na comunidade científica, que suporta a linguagem, um esforço muito relevante [Evans 00, Clark 00, Engels 00, Clark 01, Kleppe 01] para dotar o meta-modelo de uma semântica formalmente definida e, em consequência, tornar rigorosos e não ambíguos os modelos UML (ou pelo menos parte desses modelos).

3.4 Diagramas UML

Abordam-se de seguida os diagramas UML mais importantes para o trabalho em causa, de forma concisa e breve, permitindo ilustrar os aspectos mais significativos dos mesmos.

3.4.1 Diagramas de Casos de Uso

Os sistemas software não funcionam, de forma geral, sem interagirem com os seus utilizadores, sejam estes humanos ou não. Os sistemas disponibilizam ao ambiente em que estão inseridos funcionalidades que permitem a comunicação com os utilizadores. Um caso de uso, um *use case*, é a especificação do comportamento esperado do sistema, ou de uma parte dele, como resposta à invocação de uma determinada funcionalidade e engloba um conjunto de sequências de acções que o sistema deve executar.

Um caso de uso representa assim uma interacção entre um utilizador e o sistema [Jacobson 92]. A identificação dos casos de uso é muito importante na medida em que desta forma se recolhem os requisitos funcionais do sistema que se vai modelar. Uma mais-valia deste processo de recolha de requisitos tem a ver com o facto, decisivo, de que é nesta fase que o cliente, ou utilizadores, mais intervêm no processo. É sintomático que a captura de requisitos seja feita utilizando o vocabulário do utilizador, permitindo assim diminuir a distância semântica entre a equipa de projecto e os seus conceitos, e a visão do sistema fornecida por parte de quem o vai utilizar.

Os casos de uso de um sistema são obtidos através de uma decomposição funcional do comportamento do sistema, desde a funcionalidade de índole mais genérica até um nível mais concreto. Os diagramas de casos de uso são de capital importância, uma vez que é baseado neles que se processam as fases seguintes do processo de desenvolvimento. Toda a conceptualização que se possa fazer de um sistema é assente nos casos de uso que foram identificados durante a fase de análise.

Um diagrama de casos de uso, representa a explicitação dos casos de uso identificados e permite também associar os actores envolvidos, a sua natureza e as funcionalidades que invocam. Segundo Fowler [Fowler 97], um diagrama de casos de uso pode ser definido como:

> "the use case view captures the behaviour of a system, subsystem, or class as it appears to an outside user. It partitions the system functionality into transactions meaningful to actors - idealized users of a system.

> The pieces of interactive functionality are called use cases.
>
> A use case describes an interaction with actors as a sequence of messages between the system and one or more actors."

Os diagramas de casos de uso têm os seguintes elementos constituintes: i) os casos de uso, ii) os actores e iii) as relações existentes entre eles. Os casos de uso, representados graficamente como sendo elipses, denotam as funcionalidades do sistema e os actores, que são representados iconograficamente por um boneco de figura humana, indicam os papéis que os utilizadores do sistema desempenham. As relações são de índole de dependência, generalização e associação entre os conceitos (símbolos) presentes no diagrama.

A Figura 3.2 apresenta um diagrama de casos de uso para um sistema que permite a um aluno inscrever-se numa das conferências existentes para o efeito na sua universidade. É de notar que na fase de recolha de requisitos é natural que os casos de uso sejam associados a acções, daí o facto de corresponderem, grosso modo, aos verbos das frases que descrevem o sistema. No caso da modelação apresentada é sugestivo que os nomes dos casos de uso sejam da forma *"Procurar conferências"*, *"Inserir conferências"*, etc.

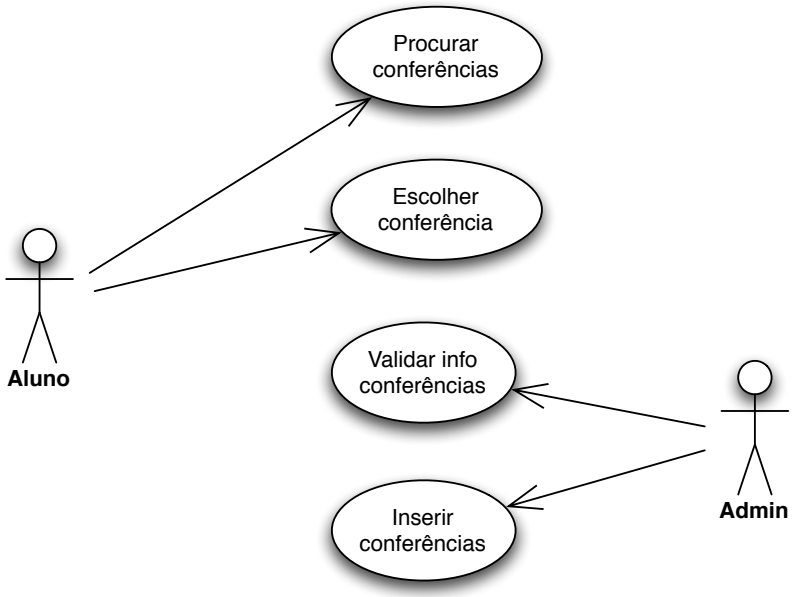


Figura 3.2: Exemplo de um diagrama de caso de uso.

Os diagramas de casos de uso modelam a face estática de uma vista, na forma de casos de uso do sistema. Permite-se assim identificar o contexto do sistema determinando os actores que pertencem ao sistema ou sub-sistema e aqueles que lhe são

externos. Por vezes, esta definição obriga a que diagramaticamente se circunscrevam os casos de uso com uma linha que delimita o sistema do contexto onde este se insere. A essa definição dá-se o nome de *diagrama de contexto*. Na modelação do contexto do sistema é possível também estabelecer relações entre os diversos actores que possibilitam a sua organização, hierárquica ou não, mas que é importante recolher durante a fase de análise.

Um caso de uso especifica o comportamento do sistema quando sofre determinada interacção e é representado por uma elipse que determina uma funcionalidade. Seja um caso de uso onde um actor desencadeia a funcionalidade de levantar dinheiro numa máquina ATM (vulgo multibanco). Do ponto de vista do diagrama de casos de uso, esse caso de uso pode ser dividido em casos de uso mais simples, como mecanismo de decomposição de complexidade, e poderíamos assim ter novos casos de uso, como *"Validar PIN"*, *"Verificar Saldo"*, etc.

Mesmo nestes casos de uso mais simples, é possível, durante a fase de análise, determinar o comportamento desta interacção. Como especificado em [Booch 05], é possível e necessário nestas situações descrever, em língua natural, esse caso de uso, recorrendo a uma descrição do tipo:

Fluxo normal:	Começa por se inserir o cartão. É pedido o PIN ao utilizador. O utilizador insere o PIN através do teclado existente na máquina. O utilizador insere os dígitos e no fim confirma com a tecla "CONTINUAR". O sistema verifica se o PIN está correcto. Se assim suceder o caso de uso termina.
Fluxo de excepção:	O utilizador pode, a qualquer momento, cancelar a introdução do PIN. Nessa situação o caso de uso volta ao princípio.
Fluxo de excepção:	O utilizador pode, a qualquer momento, corrigir o PIN que está a introduzir. Nesse caso é-lhe voltado a solicitar a introdução do PIN.
Fluxo de excepção:	Se o utilizador introduz um PIN errado o caso de uso volta ao princípio. Se isto acontecer três vezes o sistema cancela o pedido e ele não pode voltar a introduzir esse PIN.

Em situações em que num único diagrama não consta toda a informação necessária sobre as diversas alternativas, é necessário criar a noção de *cenário*. Um cenário define uma sequência de eventos originados a partir de um caso de uso, isto é, um caminho dentro desse caso de uso. De acordo com a norma [Fowler 04, Booch 05] é preferível ter diagramas diferentes para cenários diferentes, por incapacidade de representação de toda a informação no mesmo diagrama de casos de uso. Os cenários identificados

são modelados na UML, de acordo com a proposta defendida, através do recurso a diagramas de sequência, uma vez que é necessário modelar a componente dinâmica do sistema que descreve o fluxo de eventos[3].

Casos de Uso

Os casos de uso podem ser agrupados de uma forma lógica. Por exemplo, no sistema de gestão de conferências (atrás apresentado) é possível ter os casos de uso associados a funcionalidades sobre alunos agrupados num diagrama, e os relativos às conferências num outro diagrama. Este agrupamento é uma forma de organização que pode ser aplicada nas relações de generalização, inclusão e extensão previstas na UML.

A *generalização* entre casos de uso é similar à generalização entre classes. Ocorre quando um caso de uso herda o comportamento e significado do caso de uso "pai", podendo redefinir ou adicionar comportamento. Por exemplo, um caso de uso de verificação de um utilizador, de forma a efectuar o *login* no sistema, poderá ter como subcasos de uso *"Efectuar scan óptico"*, *"Verificar password"* ou *"Efectuar scan biométrico"*. Estas três formas pertencem à mesma família de operações e podem estar relacionadas no diagrama expressando essa relação. A generalização entre casos de uso denota-se graficamente da mesma forma que a generalização entre classes, ou seja, através de uma linha com uma seta na extremidade que aponta para o super-caso de uso, tal como expresso na Figura 3.3.

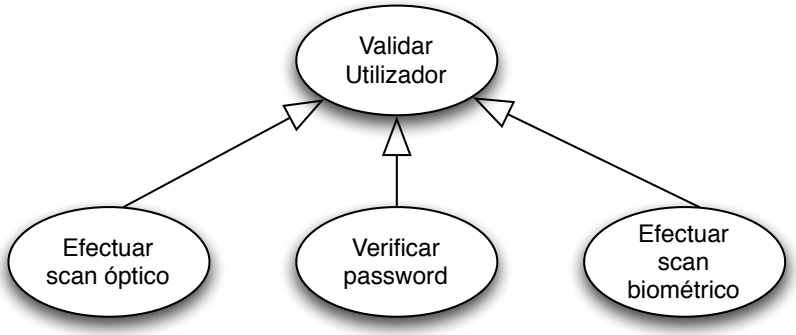


Figura 3.3: Generalização entre casos de uso.

Mais interessantes são as relações de inclusão e extensão, identificadas nos diagramas pelos estereótipos `<<extend>>` e `<<include>>` [Fowler 04, Övergaard 98].

A relação de << *extend* >> indica que o caso de uso de onde parte a seta, ou seja o caso de uso que estende a funcionalidade, pode ser utilizado pelo caso de uso

[3]Embora esta fase só ocorra muito mais tarde no processo de desenvolvimento.

estendido, num determinado momento, para efectuar uma determinada acção. Note-se que os dois casos de uso são autónomos e podem ser invocados separadamente pelos actores. Tenha-se em atenção a Figura 3.4 em que o caso de uso *"Requisitar passaporte"* representa toda a funcionalidade relativa à emissão de um documento deste tipo, encarregando-se da recolha dos dados e demais informação que envolva a emissão de um passaporte. Caso o utilizador pretenda que um passaporte lhe seja entregue com carácter de urgência, então o caso de uso *"Requisitar documentos com urgência"* é executado. A relação de << *extend* >> pode ser utilizada para modelar diversos fluxos de eventos que podem ser desencadeados por um utilizador, apesar de não ser sempre de fácil compreensão para quem lê o diagrama de casos de uso.

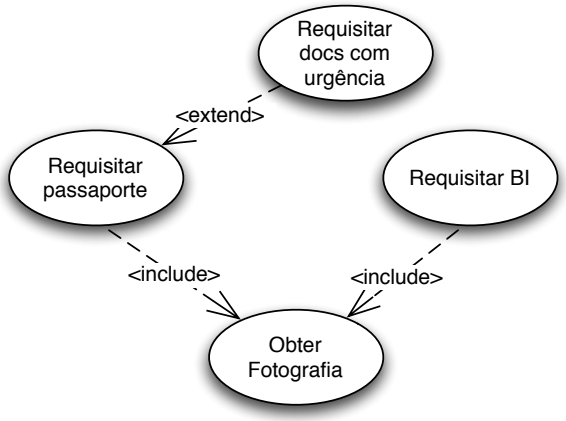


Figura 3.4: Exemplo de `<<extend>>` e `<<include>>` entre casos de uso.

A relação de inclusão, << *include* >> é utilizada quando no contexto de um caso de uso se utiliza um outro caso de uso - o apontado pela seta. A relação de inclusão é essencialmente uma relação de delegação, uma vez que o caso de uso que inclui outro, assume que este tem o comportamento correcto para o que deve ser feito. A relação de << *include* >> favorece a reutilização de casos de uso, na medida em que é possível identificar funcionalidades no sistema que sejam utilizados por outros, e modelá-las como caso de uso que pode ser incluído noutros. A Figura 3.4 mostra-se também um exemplo deste relacionamento em que o caso de uso *"Obter fotografia"* pode ser utilizado por casos de uso que necessitem daquele comportamento.

Actores

Um actor representa um papel (*"a role"*) que um determinado grupo de utilizadores podem desempenhar na interacção com o sistema. Os casos de uso são realizados por actores, podendo estes ser humanos ou não, e um mesmo actor pode invocar, ou estar

envolvido, em vários casos de uso. Um utilizador do sistema pode estar representado no sistema por mais do que um actor e um actor pode representar mais do que um utilizador. Genericamente poderíamos dizer que um actor corresponde a um perfil de utilização do sistema, sendo que um mesmo utilizador pode ser um utilizador normal do sistema, mas em determinada circunstância pode também ser administrador do sistema (se para isso tiver privilégios). Tal significa que um utilizador humano será representado no sistema por dois actores. Um actor é pois uma representação abstracta de um perfil, um tipo, de utilizadores que interagem com o sistema.

Os actores podem não ser humanos, podem ser processos ou mesmo outros sistemas que interagem com aquele que estamos a modelar.

A identificação dos actores é uma tarefa importante na definição dos requisitos do sistema feito através da definição dos casos de uso. Como um caso de uso corresponde a uma sequência de eventos que é desencadeada por um actor e descreve a interacção existente entre esse actor, o sistema e outros actores, é de importância fulcral para a definição dos requisitos do sistema. A definição de quem são as entidades que interagem com o sistema permite construir o diagrama de casos de uso e, iterativamente enunciar os casos de uso existentes.

Do ponto de vista do processo de construção do diagrama de casos de uso para especificar os requisitos do sistema, é possível enumerar um conjunto de passos que o definem:

- estabelecer o contexto do sistema através da definição dos actores envolvidos (que utilizam o sistema);
- para cada actor é necessário determinar o comportamento que cada um espera do sistema;
- identificar esses comportamentos como casos de uso;
- decompor os casos de uso descobertos de forma a descobrir comportamentos comuns;
- modelar os casos de uso descobertos, os actores e as relações entre eles num diagrama de casos de uso, e
- complementar os casos de uso com anotações que descrevem os aspectos relevantes do sistema.

Este último item não permite que, sem extensões ao modelo, os diagramas de caso de uso possam ser utilizados em técnicas de geração de código e de engenharia reversa. Enquanto que para os diagramas de classes, de componentes, de estados e de sequência é possível ter mecanismos de geração automática de código ou, inversamente, de construção de diagramas a partir de código, tal não é possível para os diagramas de casos de uso. Essa impossibilidade resulta do facto de a descrição da parte comportamental estar expressa apenas em termos de linguagem natural.

3.4.2 Diagramas de Classe

Os diagramas de classe existentes em UML são semelhantes aos diagramas encontrados em todas as metodologias orientadas aos objectos. O conceito de classe é um conceito central no paradigma e qualquer metodologia orientada aos objectos privilegia esta vista pois está muito próxima dos conceitos do paradigma e das próprias linguagens de programação.

O diagrama de classes ilustra a componente estrutural do sistema e identifica claramente as classes, interfaces e respectivas relações existentes no sistema. Os diagramas de classe são também um mecanismo necessário para a criação de outros diagramas, como sejam os de componentes e de implementação (*deployment*).

Como mecanismo de adequação ao paradigma, é o diagrama ideal para representar os conceitos, classes e tipos de dados da estrutura estática do sistema. Nestes diagramas é explicitado, à luz dos conceitos elencados, o domínio da aplicação, mesmo que não se saiba como é que esses conceitos serão implementados, embora na maioria das situações o mapeamento para uma linguagem orientada aos objectos seja quase directo.

Além de representar os conceitos, os diagramas de classe permitem estabelecer as relações existentes entre as classes. Para cada classe é possível detalhar no diagrama o estado e comportamento da mesma, isto é, as variáveis internas e a assinatura dos métodos a que cada classe responde. Esta informação pode não estar toda presente no mesmo diagrama. É normal que em sistemas mais complexos se crie um diagrama com as relações entre as classes e depois se criem diagramas mais detalhados, com a informação de cada classe, para agrupamentos de classes existentes no diagrama mais abrangente.

Em UML existem genericamente cinco tipos diferentes de relacionamento entre as classes. É possível encontrar na literatura mais formas de relacionamento, mas são variantes das que a seguir se apresentam.

As relações entre classes, no diagrama de classes, são:

1. **Associação** - uma associação entre duas classes, ilustrada por uma linha que liga duas classes, representa uma relação entre objectos dessas classes, que se manifesta em tempo de execução pela troca de mensagens. Sempre que um objecto usa as operações, os serviços, de um outro objecto deve-se sinalizar essa relação no diagrama de classes.

   A associação pode ter um nome que a identifica e caso não seja dito nada em contrário é bidireccional, ou seja, os objectos de cada uma das classes invocam serviços na outra. Se a relação não for bidireccional tal deverá ser identificado por um triângulo indicando o sentido da associação, isto é, a sua navegabilidade.

2. **Agregação** - representa a associação existente quando um objecto contém outros. À classe incluída dá-se o nome de componente e à classe que a inclui designa-se por composta, ou contentor.

   Na notação UML a agregação é representada por uma linha com um losango na extremidade do objecto composto. Na agregação não existe nenhuma obrigatoriedade de os objectos incluídos não poderem ser partilhados e referenciados por outros objectos. A agregação não garante o encapsulamento do objecto composto.

3. **Composição** - a composição é uma forma especial de agregação, com a restrição de que os objectos componentes pertencem, de facto, ao objecto composto. Graficamente representa-se por um losango cheio e significa que esses objectos pertencem ao objecto composto e não podem ser partilhados por outros objectos.

   O tempo de vida dos objectos componentes é completamente controlado pelo objecto composto.

4. **Generalização** - representa relações do tipo *is-a*, logo utiliza-se quando uma classe é uma especialização de outra. Segundo o paradigma dos objectos a subclasse herda todas as definições da superclasse, podendo adicionar ou redefinir informação e comportamento.

   Em UML essa relação denota-se por uma seta que começa na subclasse e termina na superclasse.

5. **Dependência** - a relação de dependência é utilizada para descrever as situações em que uma classe depende de outra. Essa dependência especifica que alterações à entidade destino da dependência fazem-se sentir na entidade que dela depende.

Um exemplo de uma situação em que faz sentido aplicar uma relação de dependência é descrição da relação com uma classe que é passada por parâmetro. Uma dependência é uma associação que se representa a traço interrompido.

Nas relações anteriormente apresentadas, à excepção da generalização, é possível descrever a cardinalidade dos objectos nas extremidades das relações. Essa descrição designa–se por multiplicidade e pode tomar as seguintes formas:

- *1* - uma ocorrência;
- *0..1* - zero ou uma ocorrência;
- *** - zero ou mais ocorrências;
- *1..** - uma ou mais ocorrências, e
- *m..n* - de m a n ocorrências.

A leitura do diagrama de classes faz-se pela identificação dos conceitos, classes e interfaces, e das relações entre eles com a semântica anteriormente definida. As relações podem ainda ser decoradas com informação sobre como é que devem ser lidas à luz do domínio da aplicação. É possível associar a cada extremidade de uma relação um papel (role) que ajude à leitura do diagrama.

Esta decoração das relações, fornecendo papéis, é opcional, sendo no entanto útil quando existem relações entre objectos da mesma classe ou para objectos que tenham mais do que uma relação, ajudando assim à compreensão do diagrama.

A Figura 3.5 apresenta uma súmula de um diagrama de classes com a inclusão de algumas das relações apresentadas atrás. Utilizam-se notas colocadas sobre as relações para identificar o seu tipo.

Integração no processo de desenvolvimento

Uma das razões pela qual os diagramas de classe são os diagramas UML mais utilizados prende-se com o facto de como se referiu, a distância semântica deles para as linguagens de programação não ser grande. Os conceitos e as relações entre esses conceitos são facilmente mapeados para as linguagens de programação, possibilitando que os elementos da equipa de projecto encarem de forma natural a utilização destes diagramas, na medida em que lhes permite pensar na arquitectura de classes a um nível mais abstracto, mas com a concretização em linha de vista.

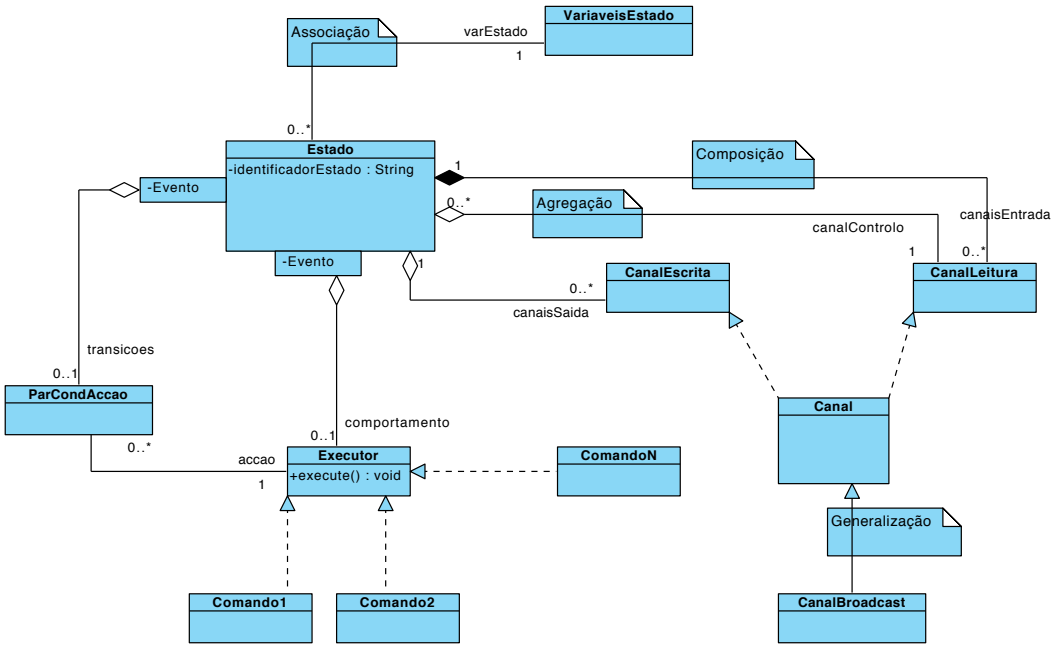


Figura 3.5: Diagrama de classes com relações de diverso tipo.

Os diagramas de classe, ao permitirem que lhes seja adicionada informação sobre o estado dos objectos, seus tipos de dados e nível de visibilidade, e sobre as operações, a sua assinatura, permitem efectuar a um nível visual o mesmo esforço que um programador faz a nível de uma linguagem de programação, apenas não descrevendo os algoritmos das operações. Desta forma, torna-se fácil ter mecanismos que dado um diagrama de classe possam gerar código numa linguagem orientada aos objectos. Para as operações, tendo em conta apenas a assinatura das mesmas, é também possível gerar o seu arquétipo, o seu esqueleto, possibilitando ao programador que descreva algoritmicamente, em código fonte, as instruções que correspondem ao comportamento desejado.

Inversamente, a partir dos ficheiros com código escrito numa linguagem orientada aos objectos, é possível reconstruir o diagrama de classes através de um mecanismo simples de inspecção de código.

O diagrama de classes oferece ainda a vantagem de permitir a sua utilização como mecanismo elementar das técnicas de engenharia reversa.É assim possível pensar no diagrama de classes como um mecanismo essencial daquilo a que usualmente se designa por *round-trip engineering*, isto é, a capacidade de a partir de um diagrama modelado

no processo de desenvolvimento gerar código final e que alterações efectuadas ao nível do código também se façam sentir a nível dos modelos diagramáticos existentes.

3.4.3 Diagramas de Actividade

Os diagramas de actividade são a técnica existente em UML para a descrição de processos de negócio e de fluxo de controlo (e dados). Têm uma notação similar à dos fluxogramas, acrescentando a capacidade de descrever acções em paralelo.

O diagrama de actividade, devido à pouca complexidade intrínseca que apresenta, a nível da notação e dos conceitos, pode ser trabalhado conjuntamente com os utilizadores de forma a que este especifique as tarefas (e respectivos fluxos) a que o sistema deve responder. Na perspectiva da utilização do diagrama, o que se especifica é uma actividade que é composta por várias acções organizadas consoante é explicitado pelo diagrama. Essa noção de actividade corresponde normalmente ao que é considerado como uma tarefa importante do sistema e cuja actividade merece ser descrita. Apesar de não existirem na linguagem mecanismos naturais para a modelação de tarefas (mais utilizados na especificação de sistemas interactivos), os diagramas de actividade podem ser utilizados com esse propósito e alterações recentes na definição da linguagem UML permitem uma maior adequação a essa descrição e propósito.

Os diagramas de actividade possibilitam a descrição das acções associadas no sistema à execução de um caso de uso, ou à conjunção de vários casos de uso para descreverem um fluxo de controlo mais alargado, a que tipicamente está associada a noção de tarefa. Do ponto de vista dos construtores gráficos existentes o diagrama de actividade é razoavelmente simples, apresentando além do estado inicial e do estado final a capacidade de descrição das acções e as variações do fluxo de controlo. No que concerne ao controlo de fluxo um diagrama pode apresentar escolhas em função do teste de condições booleanas ou a capacidade de descrição de acções em paralelo.

O comportamento condicional pode assumir a forma de uma decisão ou de um merge. Uma decisão tem uma entrada e várias saídas guardadas por expressões condicionais. Um merge tem múltiplas entradas e um único fluxo de saída, servindo desta forma para marcar o fim de um comportamento condicional.

Para a descrição de tarefas que ocorrem simultaneamente os diagramas de actividade prevêem a utilização de fork e join. Um fork tem um fluxo de entrada e vários fluxos concorrentes de saída. Uma vez que existem vários fluxos que são lançados em paralelo é necessário que eles sincronizem em determinado ponto. Um join permite que se especifique que o fluxo de saída só será efectuado quanto todos os processos

concorrentes tiverem terminado.

A Figura 3.6 representa um diagrama de actividade para uma tarefa em que uma empresa recebe uma encomenda e trata de entregar os produtos pedidos e receber o respectivo pagamento.

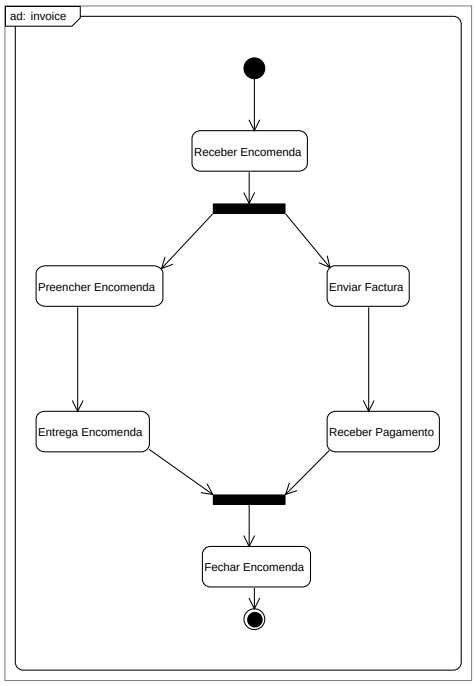


Figura 3.6: Diagrama de actividade relativo ao processamento de uma encomenda.

No diagrama especifica-se que as acções de satisfação da encomenda e o processo financeiro de emissão da factura e recebimento do dinheiro pode ocorrer em paralelo. Apenas quando estes dois fluxos terminam é que se torna possível desencadear a acção que efectua o fecho da encomenda (Fechar Encomenda) e chegar ao estado final da actividade.

Por uma questão de estruturação e gestão da complexidade, é possível num diagrama fazer referência a sub-actividades.

Neste diagrama, além das acções e das decisões, também se representam os parâmetros de entrada e saída da sub-actividade. Uma acção pode ser etiquetada, com a identificação do método que é invocado.

Essa descrição é feita da forma `nome-classe::nome-metodo`.

Na especificação de uma actividade, ao identificar os vários fluxos de execução e as acções envolvidas é possível associar responsabilidades na sua execução. Um diagrama pode ser dividido em pistas de forma a descrever quem faz o quê na actividade.

3.4.4 Diagramas de Estados

Os diagramas de estado, ou de máquinas de estado [Harel 87], são uma técnica utilizada para descrever a componente dinâmica de uma entidade. Efectuar esta descrição implica especificar como é que ao longo da vida do sistema os objectos, os casos de uso e mesmo componentes do sistema se comportam e alteram o seu estado. Tenha-se em atenção que o estado global de um sistema corresponde à conjunção de todos os estados internos de cada um dos objectos que o compõem.

Com a evolução do sistema, durante o seu ciclo de vida, os objectos respondem a eventos, a interrupções, à invocação de operações ou apenas ao passar do tempo. Quando um evento sucede é desencadeada uma actividade que pode alterar o estado interno dos objectos em causa.

Em UML os diagramas de estado são importantes na medida em que permitem fazer a descrição do comportamento dinâmico dos objectos. Esta informação não está presente nas outras vistas, nos outros diagramas, de forma directa e intuitiva. O diagrama de classes não permite aferir do comportamento dinâmico dos objectos, salvo aquele que é possível adivinhar da descrição das operações cuja assinatura é apresentada. Também é possível estabelecer algum raciocínio através das relações expressas no diagrama de classe, mas não é fácil determinar o comportamento dinâmico dos objectos nessa vista.

Justifica-se desta forma a existência de uma notação que capture os requisitos dinâmicos e que tenha subjacente um meta-modelo baseado em estados e com capacidade de descrição das transições entre esses estados. Os diagramas de estados são uma ferramenta teórica útil para modelar o comportamento dinâmico de um sistema, permitindo identificar os estados pelos quais um objecto passa e recolher todas as transições existentes entre esses mesmos estados e as condições em que as transições podem ser efectuadas. Um dos desafios na modelação do comportamento dinâmico é a correcta definição do que é um estado e quais são os estados existentes na vida de um objecto. Os estados devem ser identificáveis pela alteração que se produz internamente ao objecto e pelo período de tempo em que essas alterações se reflectem.

Segundo Douglass [Douglass 98] um estado é:

> "a state is an ontological condition that persists for a significant period of time, is distinguishable from other such conditions, and is disjoint with them. A distinguishable state means that it differs from other states in the events it accepts, the transitions it takes as a result of accepting those events, or the actions it performs. A transition is a response to an event that causes a change in state."

A modelação de um objecto numa máquina de estados facilita a compreensão da sua natureza dinâmica, mas impõe simplificações, que possibilitam essa descrição. Assume-se que:

1. o objecto a ser modelado assume um número finito de estados, identificáveis pelas condições do seu estado interno;
2. o comportamento do objecto num determinado estado é definido por
   (a) as mensagens e eventos passíveis de serem enviados
   (b) as acções associadas a cada evento
   (c) o grafo de atingibilidade possível de criar a partir do estado
   (d) os tuplos *evento,transição,próximo_estado* que se podem identificar
   (e) as acções efectuadas enquanto o objecto está num determinado estado
   (f) as acções que são efectuadas quando um objecto entra, ou sai, de um estado
3. o objecto está num determinado estado por um período temporal não negligenciável, isto é, possibilita a identificação de um comportamento que pode ser rastreado;
4. o objecto pode apenas mudar de estado através de um conjunto finito, e conhecido, de transições, e
5. as transições são acções que não demoram tempo, isto é, são instantâneas.

Os *diagramas de statecharts*, a que chamaremos por simplificação diagramas de estados, possibilitam a modelação de eventos que induzem a transição de estado. No contexto deste trabalho, interessam-nos mais os sistemas em que a mudança de estado de um objecto é desencadeado por um estímulo externo, síncrono ou assíncrono, por esta ser eminentemente a natureza dos sistemas complexos e por esta forma de modelação entroncar melhor no espírito orientado aos objectos. Por exemplo, sistemas de *workflow* são idealmente especificados recorrendo a diagramas de actividades.

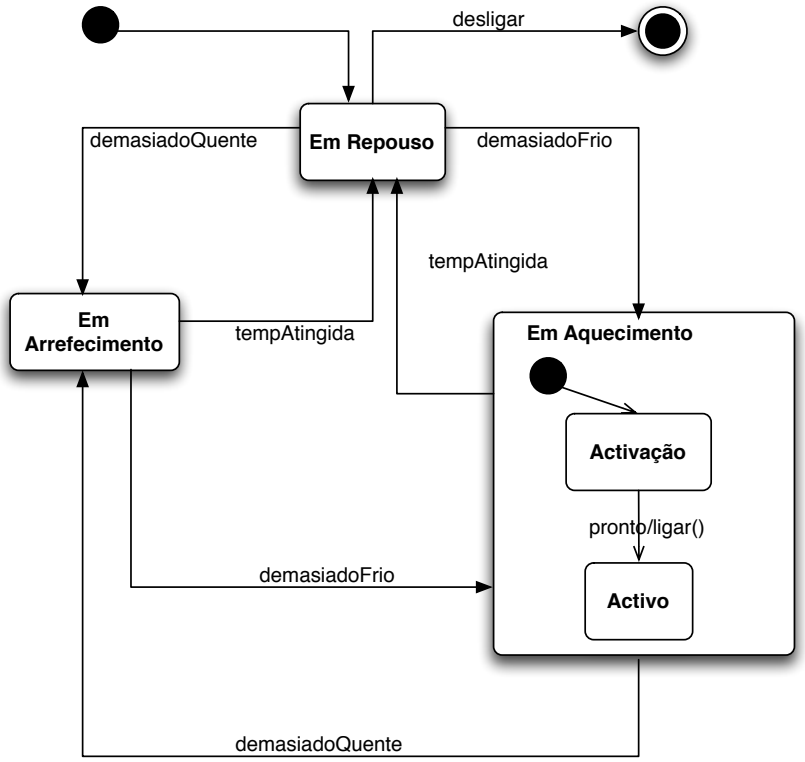


Figura 3.7: Diagrama de transição de estados para um sistema de aquecimento.

A Figura 3.7 mostra um exemplo de um diagrama de estado para um sistema de aquecimento de uma casa. Estados e transições são as entidades relevantes num diagrama de estados: os estados são representados por rectângulos com os cantos arredondados e as setas indicam as transições. Neste diagrama existe um estado de características particulares, um *super-estado*, que é um estado que contém outros no seu interior. Este é um mecanismo de estruturação de *statecharts* que permite que este super-estado possa ser reutilizado noutros contextos, abstraindo-se assim a sua complexidade. Quem escreve o diagrama apenas se preocupa com as transições que conduzem o comportamento até ao super-estado, não sendo relevante a especificação do comportamento interno deste. Existem ainda dois estados de carácter especial, o *estado inicial* e o *estado final*, que indicam respectivamente o estado inicial por omissão e o estado onde a execução termina. O estado inicial é representado por um círculo a cheio e o estado final por um círculo a cheio circunscrito por outro círculo.

No diagrama da Figura 3.7, o sistema de aquecimento está inicialmente em "Repouso"e em função da temperatura ambiente transita para os estados "Em Arrefecimento"ou "Em Aquecimento"em função do evento que tenha sido desencadeado, a

saber *demasiadoQuente* ou *demasiadoFrio*. Quando a temperatura ideal é atingida, o sistema de aquecimento volta a entrar no estado de repouso de onde também pode sair se o sistema for desligado.

As transições indicam um movimento de um estado para outro. Cada transição pode ser etiquetada, sendo essa etiqueta constituída por três partes, todas elas opcionais, que assume a forma de *Evento [Guarda] /Acção*. O *Evento* indica o nome do evento que dispara a transição, a *Guarda* corresponde a uma expressão lógica que habilita a transição[4] enquanto que a *Acção* corresponde a uma operação que é executada quando se efectua a transição. As acções podem tomar a forma de atribuições a valores de variáveis do sistema ou ao desencadear de eventos. É possível que a descrição da transição não contenha todos elementos apresentados. Se, por exemplo, a transição não contiver uma acção significa que durante essa transição não se executa nenhuma operação.

Os eventos representam uma ocorrência significativa no tempo e no espaço. Existem diversos tipos de eventos [Rumbaugh 05, Fernandes 00], como é evidente na Tabela 3.3.

Existe em UML uma diferença evidente entre *acção* e *actividade*, embora ambas representem operações que são executadas e que pertencem à lista de operações disponíveis na especificação de um objecto. Uma acção está associada a uma transição e executa-se quando o evento é desencadeado e a guarda é verdadeira. Uma actividade está explicitamente associada aos estados e é algo que o objecto executa quando o objecto esta num determinado estado[5].

Existem acções que podem ser associadas à entrada e saída de um estado, designados por *entry actions* e *exit actions*. É possível tipificar as transições e as acções implícitas a elas, como constante da Tabela 3.4:

[4]Por exemplo, só faz sentido activar o aquecimento se a temperatura estiver abaixo de um determinado valor.

[5]Na notação UML, nas versões 1.x utilizam o termo acção para as actividades normais associadas a transições e apenas se designavam por actividades aquelas que estavam associadas a entradas com prefixo do. Na versão 2.0 da linguagem a palavra actividade passa a designar ambos os casos.

Tipo de Evento	**Descrição**	**Sintaxe**
Evento Chamada (Call Event)	Representa um pedido síncrono entre objectos. O objecto que invoca o pedido fica parado (bloqueado) à espera que o receptor da mensagem responda.	op(a:T)
Evento Transição (Change Event)	Documenta a mudança de um valor booleano. Este evento não deve ser confundido com uma guarda. Uma guarda é testada para verificar se a transição pode ser efectuada, enquanto que um evento deste tipo é avaliado continuamente até que seja verdadeiro, situação em que a transição é efectuada.	when(exp)
Evento Sinal (Signal Event)	Representa a recepção explícita de uma invocação assíncrona para comunicação entre objectos. O emissor do sinal não fica à espera que o receptor processe o sinal e continua a sua actividade.	sname(a:T)
Evento Temporal (Time Event)	Regista a ocorrência de um determinado valor para o tempo absoluto ou a passagem de um determinado intervalo de tempo. Num modelo de alto nível os eventos temporais provêm do universo, enquanto que numa implementação real são obtidos de um objecto específico, quer seja do sistema operativo ou da própria aplicação.	after(time)

Tabela 3.3: Classificação dos eventos.

Tipo de actividade	**Descrição**	**Sintaxe**
entry action	A acção é executada quando se entra no estado	entry/acção
exit action	A acção é executada quando se sai do estado	exit/acção
transição externa	Caso geral de resposta a um evento que causa a mudança de estado se as condições forem verdadeiras.	e(a:T)[exp]/acção
transição interna	Caso particular do anterior, mas não implica a mudança de estado nem a execução das acções de entrada e saída.	e(a:T)[exp]/acção

Tabela 3.4: Actividades de um estado.

É possível associar à informação de um estado a descrição de actividades que ocorrem quando um objecto se encontra num determinado estado. Essas actividades correspondem a processos que são sempre executados, permitindo no entanto que a execução seja interrompida pela ocorrência de um evento. Essas actividades são descritas pela sintaxe *do/actividade*.

A Figura 3.8 apresenta um estado com várias actividades, consoante a descrição feita anteriormente.

Figura 3.8: Estado com actividades associadas.

Tendo já introduzido a noção de estado, de transição e das actividades associadas quer aos estados quer às transições, é importante nesta breve descrição abordar duas situações relevantes: a dos estados com história e a dos estados com concorrência interna.

Um *estado com história* descreve o comportamento dinâmico de um objecto cujo comportamento actual depende do seu comportamento passado. Quando uma transição atinge um estado composto, um super-estado, as acções neste estado começam no seu estado interno inicial. Se, pelo contrário, pretendermos modelar um estado em que exista memória de forma a recordar o último estado interno em que se encontrava um estado composto, é necessário recorrer a um conector história, um *shallow history state*.

Veja-se o exemplo da Figura 3.9 onde se modela o comportamento de um agente que faz cópias de segurança remotas. É necessário, caso ele seja interrompido, saber onde estava antes da interrupção. O conector com história é representado por um círculo com um H dentro e mantém informação sobre o último estado visitado. Quando se pretende retomar, num estado composto, o comportamento onde este estava pela última vez deve-se colocar a seta de transição directamente para o conector com história. No caso de ser a primeira vez que o estado composto é alcançado o conector com história aponta, por omissão, para o estado inicial interno.

O comportamento dinâmico de um estado composto é na maioria das situações modelado por estados (neste caso estados internos) que são sequenciais, isto é, uma transição leva a um novo estado que é único. No entanto é possível especificar o com-

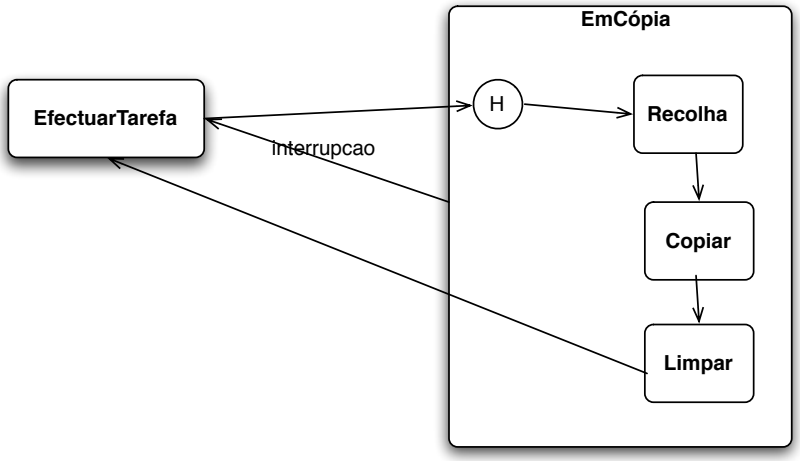


Figura 3.9: Utilização do conector história.

portamento de um estado em que duas, ou mais, máquinas de estado executam em paralelo no contexto de um objecto. Este tipo de comportamento pode ser também especificado pela criação de diferentes objectos, activos, para cada sub-estado, embora também se possa modelar como sub-estados concorrentes, de forma a descrever concorrência interna num objecto a nível de máquina de estados. Na Figura 3.10 é apresentado um exemplo de um estado com concorrência interna. A execução de dois

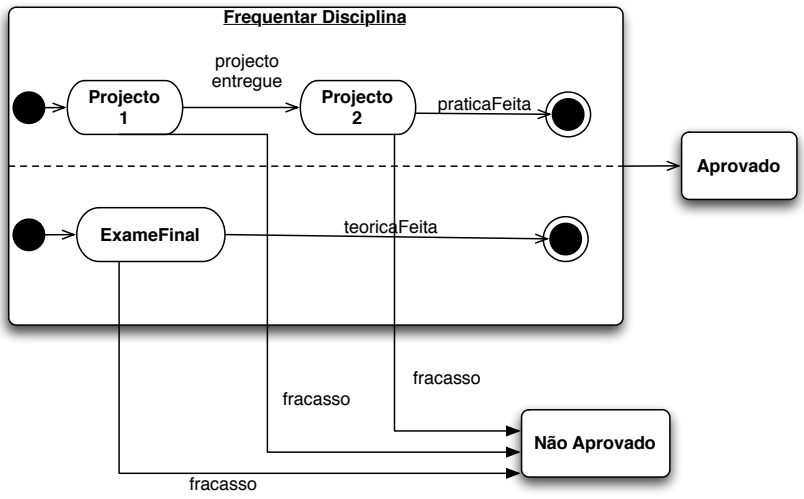


Figura 3.10: Concorrência interna a um estado.

sub-estados concorrentes ocorre em paralelo. Se uma linha de concorrência chega ao fim antes da outra, espera pela conclusão da outra linha de execução e quando ambas

terminam o controlo é unificado num único fluxo.

3.4.5 Diagramas de Sequência

Os diagramas de sequência são um caso particular dos diagramas de interacção. Estes descrevem a forma como grupos de objectos colaboram entre si de forma a obter determinado comportamento. Esse tipo de colaboração, e tendo em linha de conta que se tratam de objectos, assume a forma de troca de mensagens.

Os diagramas de estado apesar de capturarem informação sobre algumas das mensagens trocadas, só o fazem a um nível de detalhe que é o do objecto e não contém informação relativa a objectos de outras classes. É necessário descrever as trocas de mensagens e os objectos envolvidos na prossecução de um caso de uso, ou para ser mais preciso, de um cenário de um caso de uso [Fowler 04, Douglass 98].

Um diagrama de sequência apresenta uma vista da interacção num formato bidimensional, onde o eixo vertical é o eixo temporal e o eixo horizontal elenca os objectos, e respectivos papéis, presentes na interacção. Cada objecto é representado por uma linha vertical cuja dimensão no respectivo eixo representa o seu tempo de vida.

A Figura 3.11 apresenta um diagrama de sequência onde se ilustra a maioria dos conceitos presentes nestes diagramas: objectos, mensagens, períodos de inactividade e anotações textuais.

As mensagens trocadas entre os objectos são representadas por setas horizontais entre o objecto emissor e o objecto receptor. Como uma mensagem corresponde à invocação de uma operação, o nome deste é colocado junto à seta, bem assim como se identifica se é uma mensagem síncrona (a situação mais comum) ou uma mensagem assíncrona. A leitura do diagrama faz-se de cima para baixo, sendo que o eixo temporal não tem associada qualquer escala, apenas permitindo raciocinar a nível de precedência de envio de mensagens.

É possível associar mais informação como seja a informação sobre o evento que desencadeou a mensagem (permitindo a associação ao diagrama de estados), bem assim como guardas à execução de algumas mensagens. Por vezes interessa ainda explicitar a criação e destruição de objectos. Existem ainda construtores gráficos que permitem especificar comportamentos cíclicos, alternativos, paralelos, entre outros. Para a especificação destes conceitos, em UML 2.0 utiliza-se a noção de *frame de interacção*, que representa uma região cujo comportamento está sujeito ao teste de uma condição booleana.

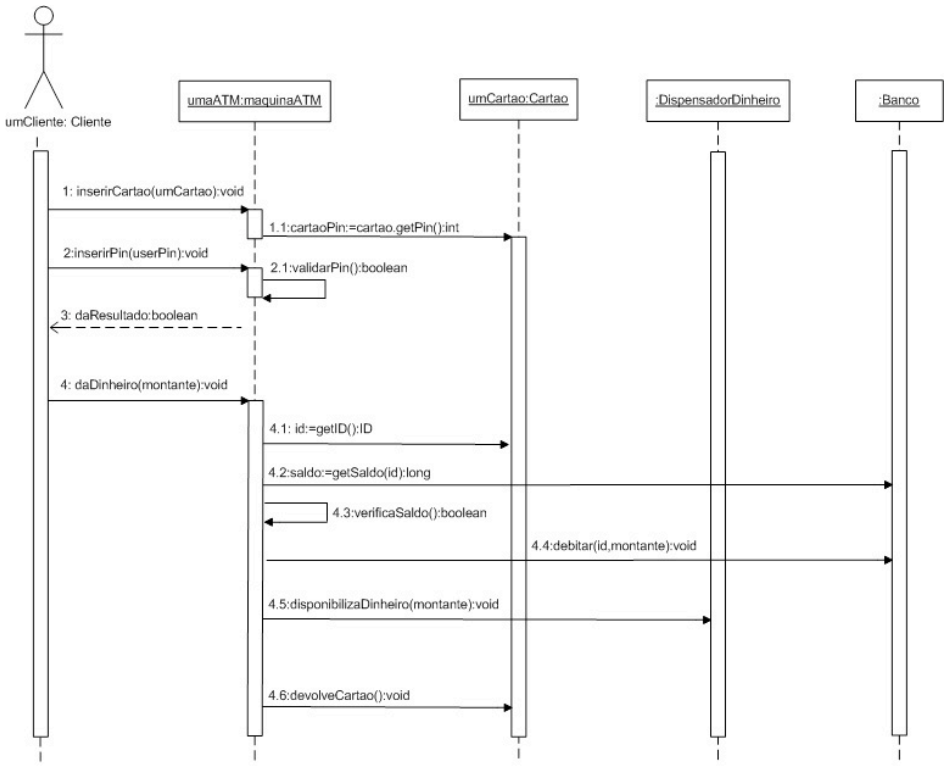


Figura 3.11: Exemplo de um diagrama de sequência.

Em resumo, os diagramas de sequência são utilizados sempre que se pretende detalhar o comportamento de vários objectos no contexto de um cenário de um caso de uso. Permitem estabelecer as características base da colaboração entre os objectos e são uma representação de alto nível do algoritmo que vai ser implementado.

3.5 A UML 2.0

Como foi referido a UML posicionou-se como uma linguagem de modelação adaptável a metodologias de desenvolvimento orientadas aos objectos e rapidamente se tornou muito popular e a reacção dos utilizadores foi muito positiva. Este efeito derivava mais da vertente unificadora de conceitos que propriamente da novidade introduzida. No entanto, teve como consequência uma valorização mais concreta da modelação, e das actividades subjacentes, especialmente no que toca a sistemas de complexidade evidente.

Na realidade a sua utilização radicou mais na sua faceta de documentação do que

numa efectiva ajuda no processo de desenvolvimento. As razões desta atitude prendem-se sobretudo com a imprecisão e ambiguidade que os modelos desenvolvidos na linguagem transportam, o que não os torna definitivamente auxiliares fiáveis no processo. Pode inclusive definir-se que a mais-valia concreta de um modelo é directamente proporcional ao grau de precisão e rigor que este apresenta. A solução parece pois, passar pela diminuição da distância entre os modelos e o sistema que estes representam. A relação formal entre um modelo e o correspondente sistema software é o objectivo pretendido, sendo que o sistema deve ser obtido através de transformações, automáticas ou não, do modelo.

Esta relação entre o modelo e a sua concretização pode ser também entendida, se o modelo de processo for formal, no sentido inverso, isto é, do sistema software para a construção do modelo. A combinação de mecanismos de abstracção e de concretização automática, entre as técnicas de modelação e os métodos de desenvolvimento associados, resultou no que actualmente se designa por desenvolvimento orientado ao modelo, *Model Driven Development* (MDD) [Booch 04]. Esta camada metodológica define como entidades de primeira ordem os modelos, dando-lhes um papel mais relevante no processo, ao mesmo tempo que retira importância à componente de desenvolvimento propriamente dita. Quanto mais precisos e rigorosos forem os modelos, maior o nível de automatismo que se consegue e maior a capacidade de abstracção em relação à componente de produção de código.

A maior importância dos modelos e a maior preponderância das normas (standards) MDD, influenciou a revisão substancial feita à notação, de forma a criar a versão 2.0 da UML. Esta versão seguiu-se a uma série de revisões menores [Group 04, Booch 05] cujos números de versões mais importantes foram a 1.3 e a 1.5.

A UML 2.0 resulta fundamentalmente da necessidade de suportar melhor as ferramentas e métodos de MDD, logo de se ter uma definição mais precisa e rigorosa da linguagem, que possibilite um nível de automatismo mais elevado. As grandes diferenças introduzidas pela versão 2.0 são [Selic 06]:

1. um aumento do rigor na definição da linguagem - eliminando a ambiguidade e imprecisão que modelos das versões anteriores da linguagem continham;

2. um aumento na organização da linguagem - pela maior modularização da mesma, facilitando a utilização por ferramentas diversas;

3. melhorias na capacidade de descrição de sistemas de larga escala - através da incorporação de novos mecanismos que permitem ter diferentes níveis de abstracção e de complexidade;

4. extensão para incorporação de informação sobre o domínio da aplicação, e

5. clarificação de vários conceitos - eliminando redundâncias e redefinindo definições de forma a providenciar uma linguagem consistente.

Aumento de rigor na definição da linguagem

Em algumas situações, e devido a deficiência na definição do meta-modelo UML, os conceitos adstritos à modelação necessitam de clarificação com recurso à utilização de linguagem natural. Embora numa primeira fase de afirmação da linguagem esta informalidade não fosse crítica, à medida que se vai utilizando a notação começam a manifestar-se sintomas que tem a ver com a ambiguidade e imprecisão dos modelos. O aspecto mais sintomático dessa ambiguidade resulta do facto de diferentes elementos da equipa de projecto interpretarem de forma diferente o mesmo modelo.

A minimização da ambiguidade faz-se pela definição de um meta-modelo, onde se defina a semântica de cada conceito da linguagem. O meta-modelo da UML foi construído recorrendo a um subconjunto da linguagem a que se adicionou a capacidade de expressar restrições escritas em OCL (Object Constraint Language). Obteve-se uma especificação formal para a sintaxe abstracta da UML, isto é, definiu-se o conjunto de regras que permitem determinar se um modelo é bem formado (se está bem escrito). Esta especificação é, puramente sintáctica, ao nível das construções que se podem desenhar, não permitindo nenhuma espécie de raciocínio ao nível da semântica do modelo. A semântica do modelo, o que representa, é de importância decisiva para que se possam associar mecanismos de geração de código ou de transformação formal do modelo.

A definição UML 2.0 pretende colmatar esta lacuna através de:

1. Uma refundação da infra-estrutura do meta-modelo - através da clarificação de padrões e conceitos da notação que ou eram muito básicos, ou então de definição muito abstracta;

2. Uma descrição mais precisa da semântica - equilibrando a definição da semântica entre as diversas vistas, diagramas, da linguagem. O esforço foi sobretudo feito nas componentes da linguagem que tinham menor suporte formal e na definição rigorosa da componente comportamental [Selic 04];

3. Uma definição clara da plataforma estrutural da linguagem - através da separação dos conceitos base da linguagem e da definição semântica dos blocos principais.

Na versão 2.0 da linguagem ficou mais clara a separação entre a componente estrutural e a comportamental e a definição do modelo causal dos comportamentos UML (actividades, máquinas de estado e interacções). A Figura 3.12 apresenta o diagrama de blocos da plataforma estrutural da linguagem.

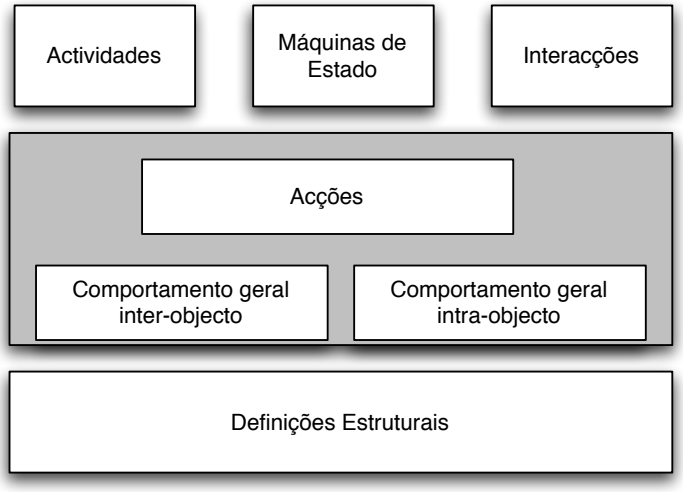


Figura 3.12: A plataforma estrutural da UML 2.0.

Nova arquitectura da linguagem

Com o esforço de incremento de formalismo a linguagem ficou mais extensa, com mais construtores, possibilitando que se descrevam mais particularidades do sistema. A definição da linguagem foi também reestruturada, através de um esforço de modularização dos diversos componentes que a compõem, sendo que como a Figura 3.13 apresenta, em cima de uma camada comum com os construtores básicos, colocaram-se a definição das diversas unidades de linguagem. Esta construção permite que possam surgir alterações ao nível destas sub-unidades de linguagem, sem que o seu impacto se reflicta na componente comum e nas outras unidades de linguagem.

Modelação de sistemas de larga escala Uma das vertentes em que a versão 2.0 da UML mais novidades trouxe foi na capacidade de descrição de sistemas de larga escala e complexos, na medida em que se acrescentaram novos diagramas ou então se acrescentou capacidade de modelação a diagramas existentes. As alterações feitas aos diagramas foram no sentido de aumentar o grau de expressividade dos mesmos, possibilitando a descrição de comportamentos complexos, tendo existido também a preocupação em tirar o máximo partido possível dos mecanismos de reutilização de modelos, ou de partes de modelos. Sendo assim, foram adicionadas capacidades de

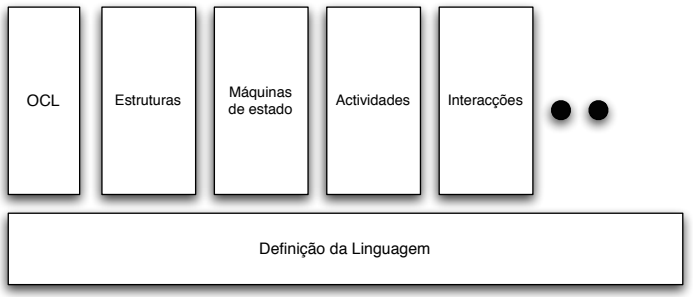


Figura 3.13: A arquitectura da linguagem UML 2.0

modelação nas seguintes vertentes:

- Estruturas complexas;
- Actividades;
- Interacções, e
- Máquinas de estado.

As *estruturas complexas*, são baseadas em conceitos de linguagens de descrição arquitectural como sejam SDL [Union 02] ou UML-RT [Douglass 98, Douglass 04]. Estes conceitos são baseados em noções derivadas da teoria de grafos e permitem especificar componentes interligados por canais de comunicação. Estes grafos podem ser constituídos como componentes reutilizáveis, de forma a permitir um grau de abstracção superior (vendo o componente como uma caixa negra).

As *actividades*, são utilizadas para modelar fluxos, sejam de dados, sinais ou de controlo. Estes diagramas constituem uma ferramenta interessante para a modelação de processos de negócio ou modelação de transformação de informação. Para suprir as deficiências da UML 1.x no que respeita a estes diagramas, basicamente por estarem restritos à semântica das máquinas de estados, na UML 2.0 a fundação teórica está baseada numa variante das redes de Petri coloridas e foram adicionados novos construtores como: i) interrupção do fluxo de actividades, ii) melhorias na estruturação do controlo de concorrência e iii) diversos modelos de comportamento.

As *interacções* tornaram-se na versão 2.0 da linguagem uma nova entidade de modelação (apesar de existirem diagramas de interacção nas versões anteriores). Uma interacção representa uma sequência de comunicações entre objectos de complexidade arbitrária, que pode ser parametrizada de forma a constituir-se como um padrão de

comportamento. As interacções em UML 2.0 podem ser representadas por diagramas de sequência e encapsulam um comportamento complexo que pode ser utilizado na descrição de outros comportamentos. O esforço efectuado foi ao nível do encapsulamento de complexidade permitindo uma capacidade de abstracção superior.

As *máquinas de estado* também sofreram alterações semelhantes às apresentadas para as interacções. Os esforços foram feitos no sentido de se permitir ter um estado composto, que fosse modular, com pontos de transição associados a *entry actions* e *exit actions*, bem definidos. Possibilita-se desta forma a reutilização destes estados estruturados noutras máquinas de estado, permitindo-se a descrição de comportamentos complexos de forma mais simples.

3.5.1 Sumário das alterações

As alterações introduzidas pela UML 2.0 foram efectuadas no sentido de providenciar maior rigor e precisão à linguagem, procurando colmatar algumas das falhas apontadas a versões anteriores. Procurou-se nesse sentido dotar a linguagem de uma fundação semântica mais bem estruturada de forma a diminuir a ambiguidade de alguns construtores e também a permitir que o processo de geração de código a partir dos modelos fosse uma realidade.

Do ponto de vista da linguagem foram introduzidos novos diagramas, embora a grande diferença surja no facto de se terem criado, a nível comportamental, entidades compostas que encerram complexidade e que permitem parametrização e reutilização noutros contextos.

A linguagem do ponto de vista da introdução de nova notação não sofre alterações substanciais, mas no que concerne à sua estruturação foi substancialmente modificada, permitindo evolução separada de cada um dos componentes, o que em princípio permite uma melhor adequação às ferramentas existentes no mercado.

3.6 Incorporação de rigor na UML

Como já foi explanado, a UML constituiu-se como uma alternativa às linguagens de modelação existentes. A incorporação de conceitos de outros métodos orientados aos objectos contribuiu muito para a sua aceitação. Ao constituir-se como uma norma devidamente certificada pelas entidades competentes (OMG), a UML passou a ser utilizada no processo de desenvolvimento de sistemas software, quer esta sua participação

seja mais orientada à documentação, ou mais orientada ao acompanhamento efectivo do processo de desenvolvimento. Neste capítulo abordam-se as situações em que a linguagem não permite suportar de forma eficiente os requisitos do processo de análise.

A falta de algum suporte semântico, faz-se sentir sobretudo em sistemas complexos e de larga-escala, nomeadamente sistemas em que a ocorrência de concorrência, distribuição, tempo real e especificação complexa de tarefas, associadas à envergadura do projecto, mais revelam as insuficiências da UML. É possível que problemas que enderecem estas particularidades, possam ser convenientemente servidos pela linguagem, seja pela sua dimensão mais reduzida, quer seja pela homogeneidade e experiência da equipa de projecto. No entanto, estes problemas de falta de capacidade de expressão, fazem-se sempre sentir quando os problemas são complexos e quando a tarefa de análise é uma tarefa importante do processo de desenvolvimento de uma solução.

Se porventura, a tarefa que a equipa de engenheiros de software tem em mãos for de índole estrutural, já ao nível da concepção arquitectural, é perfeitamente aceitável que a utilização da linguagem, e perfeita adequação da mesma, não ofereça tantas reservas. Quando se alarga o processo de desenvolvimento de um sistema a todas as fases previstas na metodologia que se utiliza, é expectável que os problemas de utilização da mesma ocorram com mais frequência.

A maior limitação da utilização da UML surge quando o projecto em causa é não-trivial. Nessas situações a falta de semântica rigorosa, não ambígua, de algumas das suas ferramentas propicia que a informação produzida não possa ser considerada como uma boa fundação para as fases subsequentes do processo de desenvolvimento. No entanto, o maior problema parece ser aquele que determina alguma incapacidade para a descrição das necessidades de modelação dos sistemas não triviais e que não são puramente sequenciais, com algoritmos simples, e centralizados. A falta de uma semântica claramente definida para todos os elementos da notação, origina que:

- A correcta interpretação dos diagramas produzidos pode ser mais aparente do que real. Existem diagramas que são facilmente compreendidos, por mapearem conceitos bem estabelecidos e próximos das linguagens de programação, mas existem vistas que necessitam de uma semântica coerente de forma a serem consistentemente bem interpretadas. Numa equipa de projecto, o facto de diferentes elementos da equipa interpretarem de forma diferente uma mesma vista, revela da ambiguidade existente na linguagem;

- A inexistência de uma visão unificada sobre a interpretação de uma determinada peça de modelação, origina que o processo de desenvolvimento sofra o impacto

do tempo gasto em compensar a ambiguidade existente. Recorre-se a descrições em notações informais para complementar a informação expressa, logo inibindo a evolução da documentação ao nível do modelo, originando um outro nível de imprecisão e falta de rigor. Informação relativa ao tipo de aplicações atrás referido, é muitas vezes não explicitada, sendo necessário recorrer a elementos de documentação auxiliares para descrever esta informação.

Esta característica induz que um modelo de um sistema pode conter ilhas de ambiguidade e falta de rigor, por insuficiência da linguagem em que o modelo está escrito;

- Não é trivial efectuar uma análise semântica rigorosa de modelos, na medida em que estes apenas podem ser validados informalmente. É possível validar um modelo na sua componente sintáctica e estrutural, isto é, se este respeita a sintaxe abstracta definida (a Abstract Syntax da UML). Desta forma torna-se possível perceber se todas as vistas do modelo se referem a entidades existentes, verificar se as mensagens trocadas estão efectivamente declaradas, entre outras validações.

No entanto, em relação à questão que determina se o modelo global é semanticamente coerente, a resposta é muito mais difusa, proporcionando-se alguma ambiguidade.

Apesar da abrangência de que a linguagem beneficia, não foi concebida de raiz para poder incorporar técnicas formais e rigorosas na sua definição, de forma a poder raciocinar sobre os modelos e efectuar tarefas de verificação sobre os mesmos. A única incorporação de métodos rigorosos é efectuada ao nível da utilização da Object Constraint Language (OCL), cujas frases podem ser verificadas a nível semântico. A falta de formalismo na definição da linguagem, e porventura, o tipo de ferramentas utilizadas pelas equipas de projectos, levaram a que durante muito tempo a UML dispensasse as técnicas formais para certificarem rigorosamente os modelos. Entre essas causas está decerto o facto de a distância entre os diagramas UML, que recorrem a símbolos gráficos, e as notações textuais em que se baseiam as técnicas formais ser significativa e muitas vezes inibidora para elementos da equipa de projecto. No entanto, é de realçar que existem notações gráficas na linguagem, para as quais existe fundamento teórico bem estabelecido, como sejam os diagramas de máquinas de estado, possibilitando que sejam efectuadas tarefas de verificação e simulação de comportamento a um nível formal.

Genericamente pode considerar-se que ter incorporada na notação UML técnicas de especificação formais permitiria ter:

1. mecanismos precisos para especificar as regras de manipulação e composição de componentes nos diagramas;

2. forma de provar propriedades acerca das entidades representadas, como sejam invariantes e outras, e

3. ferramentas de verificação de concordância e completude entre os diversos componentes da linguagem.

O facto de o meta-modelo da linguagem ser descrito utilizando a própria linguagem UML dificulta a incorporação de novos mecanismos, que introduzam construtores formais com o objectivo de definir a semântica dos diagramas. Esta lacuna da linguagem faz-se sentir mais na sua componente comportamental. É possível ter modelos estáticos e estruturais muito similares para sistemas muito diferentes ao nível do seu comportamento. Fundamentalmente ao nível da componente de análise onde os requisitos funcionais, e de outras índoles, são identificados, o diagrama de casos de uso não está suportado por nenhum formalismo evidente, sendo que constitui-se como a peça da linguagem que manifesta mais incapacidade para capturar de forma conveniente as características do sistema que se está a modelar. Acresce o facto de ser das primeiras peças da linguagem a ser utilizada e estar, em termos de processo de software, na fase inicial do mesmo.

3.6.1 A Object Constraint Language

A Object Constraint Language, OCL, é uma linguagem formal baseada na teoria de conjuntos e em lógica de primeira ordem. A OCL é utilizada para adicionar expressões que acrescentam rigor e maior precisão às capacidades gráficas dos diagramas. Usualmente estas expressões são associadas aos diagramas de classe visto serem importantes para descrever restrições às associações entre classes, embora em teoria possam ser utilizadas em qualquer dos diagramas UML.

As restrições são na maioria das vezes escritas em linguagem natural, sendo que os próprios autores da linguagem advogam a utilização de anotações e formas organizadas de escrita de texto, para capturar essa informação. A desvantagem da escrita desta informação, importante para perceber o domínio do problema, é que a linguagem natural é uma fonte de ambiguidade e de falta de rigor, justificando a existência de uma linguagem formal para a captura dessa informação. A OCL é uma linguagem formal que devido à sua simplicidade possibilita que mesmo pessoas com poucos conhecimentos

de linguagens formais a possam utilizar[6].

A utilização normal da OCL nos modelos UML é associada à especificação do invariante do sistema e à descrição de pré e pós-condições. O invariante do sistema é descrito no diagrama de classes enquanto as pré e pós-condições são associadas às operações (ou métodos). A OCL é uma linguagem declarativa e as expressões em OCL não produzem efeitos laterais, isto é, não alteram o estado da entidade correspondente. Além destas utilizações a OCL presta-se a ser também utilizada como uma linguagem de interrogação, como mecanismos de especificação dos invariantes de tipos de dados para os estereótipos, para descrever condições em guardas, para especificar restrições em operações e para descrever as regras de transformação de atributos entre modelos UML. Não é uma linguagem executável, isto é, não existe um suporte nativo para as expressões em OCL[7], servindo como mecanismo de registo de informação sobre o modelo.

Encontram-se disponíveis em OCL tipos elementares de dados como Boolean, Integer, etc. e colecções como Set, Bag ou Sequence. Estes tipos de dados têm operações definidas sobre eles. É possível utilizar outros tipos de dados, definidos pela equipa de projecto, nos diagramas UML.

Contexto

Cada expressão OCL é definida no âmbito do contexto de uma instância de um determinado tipo de objectos. Como mecanismo de identificação do objecto em causa, o identificador `self` refere o objecto que é receptor do método. O contexto é indicado no princípio de uma expressão OCL através do prefixo `context`.

Tipos de dados

As expressões OCL são escritas no contexto de um determinado modelo UML. Todos os tipos do modelo UML podem ser utilizados na escrita de expressões em OCL. Além dos tipos de dados primitivos como Boolean, Integer, Real, String, em OCL existe um conjunto de tipos de dados que representam colecções. Em OCL `Collection` é o super-tipo das colecções como sejam Set, Bag ou Sequence. Para cada uma destas colecções são disponibilizadas operações de acesso, sendo que as operações sobre co-

[6]Originariamente foi criada na IBM como uma linguagem para descrição de processos de negócio.

[7]Embora seja possível exportar a informação das expressões e criar ambientes de execução da mesma.

lecções não alteram a colecção, apenas seleccionam algo da colecção ou resultam numa nova colecção.

Invariantes

Uma expressão OCL pode ser parte de um invariante, que é uma expressão com o estereótipo de << *invariant* >>. Quando o invariante está associado a um tipo de entidade, implica que essa expressão deve ser válida para todas as instâncias em qualquer momento. Um invariante é uma expressão do tipo booleano, ou seja em OCL, Boolean.

Por exemplo a expressão

```
context t : TurmaAlunos inv:
 t.numeroAlunos > 20
```

expressa o invariante relativo ao número mínimo de alunos que uma turma deve ter para poder funcionar.

Pré e pós-condições

Em OCL é possível escrever pré e pós-condições para uma operação. As pós-condições são avaliadas após a execução da operação e a avaliação é instantânea. Os valores das propriedades antes da execução da operação são avaliados pelas pré-condições. Para uma pós-condição se referir ao valor de uma determinada variável no início da operação tem de colocar o sufixo `pre`. Por exemplo para actualizar o saldo de uma conta bancária após um levantamento, acede-se ao valor do saldo no início da operação através de `saldo@pre`. A operação poderia ser especificada como

```
context ContaBancaria::levantarDinheiro(montante:Integer):void
pre: self.saldo > montante
post: self.saldo = saldo@pre - montante
```

Acesso a propriedades dos objectos

As expressões em OCL podem referir-se a tipos de dados, classes, interfaces e a todas as propriedades dos objectos. Como propriedades podem-se referir os atributos

do objecto, as associações e navegabilidade a partir de uma determinada classe, um método ou operação. Para cada uma destas existem na documentação, em detalhe, as particularidades da linguagem no que respeita a cada forma de acesso à informação da associação [Group 06].

A descrição dos construtores da linguagem encontra-se no Anexo 1, por ser demasiado extensa para se colocar neste capítulo.

3.7 A importância da análise de requisitos

A fase de análise é determinante no processo de desenvolvimento de um sistema software. Da fase de análise retiram-se os requisitos funcionais e não funcionais, bem como se determina o que o sistema é, em termos da sua estrutura e de como se comportará em funcionamento. A qualidade do produto final depende em boa medida da qualidade da fase de análise e da capacidade que esta tenha de apreender as características do mesmo.

Ao nível da captura dos requisitos é muito importante a capacidade de abstracção para referir o sistema software que se está a construir. A abstracção é uma técnica fundamental que quem desenvolve as tarefas de modelação pode utilizar, de forma a concentrar-se nos aspectos relevantes para a fase de modelação em que se encontra. A capacidade de abstracção ao nível da identificação dos casos de uso é muito relevante, na medida em que permite ao analista ignorar a complexidade intrínseca do sistema e focar-se apenas na componente de interacção do sistema com a envolvente. O nível de abstracção que o modelador deve utilizar depende, em boa medida, do sistema e dos interlocutores com quem tenha de trabalhar, sendo normal que sistemas mais complexos levem à existência de mais níveis de abstracção. Tipicamente na fase de captura de requisitos, e correspondente definição e descrição dos casos de uso, encontram-se pelo menos como patamares de abstracção:

- Modelo de domínio (ou de negócio) - que descreve as entidades do contexto (*habitat*) em que o sistema deve ser implementado. Um modelo com este foco é importante para estabelecer algumas regras acerca dos objectos com que o sistema vai funcionar e da forma como os utilizadores vêem o mundo das entidades que populam o sistema. A definição do domínio permite que se ganhe conhecimento, o que possibilita a elaboração da definição do relacionamento entre as entidades, com o consequente contributo para a definição do invariante de estado e das pré e pós-condições das operações;

- Especificação dos requisitos - onde se descreve o que o sistema deve fazer, sem que se tenha que detalhar como é que é feito. Esta especificação é crucial para a análise de um sistema, na medida em que permite o isolamento e descrição a um nível elevado de abstracção das funcionalidades cruciais do sistema, tal como são perceptíveis do exterior.

A especificação de requisitos do sistema deve reflectir de forma estreita o modelo de negócio. A captura de todas as especificações permite a construção de um modelo estático do sistema, de forma a criar as colaborações entre objectos que permitem animar os diversos casos de uso. O modelo estático do sistema é determinante para complementar a descrição do comportamento exteriormente exibido pelo sistema e que está maioritariamente disponível no diagrama de casos de uso. O modelo estático complementa a captura de requisitos, ao permitir associar restrições e invariantes que decorram da organização física dos conceitos. No entanto, pode afirmar-se que o conhecimento do modelo do domínio visível ao nível dos diagramas de casos de uso é fundamental para que se possa, com a ajuda dos utilizadores, especificar as restrições existentes e a função que determina o invariante do sistema, sem que se tenha de ter conhecimento profundo sobre a forma como o sistema vai ser implementado. Uma das diferenças visíveis entre a fase de especificação dos requisitos e a fase de desenvolvimento reside no facto de que, esta última define como é que os objectos do sistema se comportam no âmbito das colaborações em que intervêm, enquanto a especificação de requisitos preocupa-se apenas com a descrição dos eventos que o sistema exteriormente apresenta.

De um ponto de vista do processo de modelação, podem-se considerar [d'Souza 99] três níveis que importa reter: o domínio do problema, a especificação dos requisitos e o desenho arquitectural do sistema. A especificação dos requisitos corresponde ao comportamento externamente identificável e o desenho arquitectural versa a estrutura interna do sistema e o comportamento das entidades computacionais existentes.

O modelo de domínio aborda todos os conceitos que são importantes para os utilizadores do sistema. Na definição do modelo é necessário que se deva ter em conta a terminologia a utilizar e a percepção dos processos de negócio, relacionamentos e colaborações entre as entidades (*business entities*). A especificação dos requisitos define as principais fronteiras do sistema, ao identificar o âmbito e as responsabilidades dos principais componentes e ao detalhar os principais casos de uso e as interfaces com o utilizador. A definição do desenho arquitectural, corresponde à criação do modelo que sustenta a fase de concepção do sistema e as entidades envolvidas correspondem aos artefactos estruturais do sistema.

As tarefas de validação e verificação da componente de análise, devem ser feitas ao nível da modelação do domínio do problema e da especificação dos requisitos, na medida em que nessas fases do processo de análise o cliente (e os utilizadores) ainda está presente.

O papel do cliente, ou dos utilizadores, na construção de um sistema software é reconhecidamente muito importante, mas limitado no espectro de contribuição que pode fornecer ao analista. Tal assumpção significa que os utilizadores são muito relevantes para as tarefas de análise e modelação, mas que o seu contributo é especialmente relevante na:

- identificação dos requisitos funcionais - onde os utilizadores definem quais são, de um ponto de vista exterior ao sistema, os comportamentos expectáveis do mesmo. Nesta fase os utilizadores, devido ao conhecimento que possuem do domínio do problema e do negócio podem explicitar quais as condições para que um determinado caso de uso possa ser invocado e o que externamente é perceptível ao nível da mudança do estado do sistema. Nesta fase, é também possível aos utilizadores a descrição dos cenários que podem ser considerados ao nível da utilização do sistema. Um cenário corresponde a um conjunto ordenado de interacções com o sistema com um determinado objectivo. Ao nível do que é visível de um ponto de vista externo, a descrição que os utilizadores fazem dos cenários, corresponde maioritariamente à descrição das tarefas que podem ser efectuadas sobre o sistema;

- definição da camada interactiva - a definição da camada interactiva e o modo como as tarefas identificadas são concretizadas, é uma parte da análise e modelação que pode ser feita com a ajuda e intervenção dos utilizadores. Os cenários de utilização podem ser descritos recorrendo a técnicas mais ou menos estabelecidas, uma vez que é possível para a descrição das tarefas a utilização dos diagramas de actividade para a formalização da interacção e sequenciação das acções e do diagrama de estado para a descoberta das operações que são perceptíveis a nível da camada de interacção com o utilizador;

- validação e verificação dos requisitos - a validação dos requisitos é o processo pelo qual se determina se os requisitos estabelecidos para um sistema, ou um seu componente, estão completa e correctamente especificados. Esta validação, em última análise, depende do resultado da execução do componente (ou componentes) que implementa o requisito, bem assim como das condições impostas pelas invocações anteriores no contexto do mesmo fio de execução. A verificação faz-se

na medida em que por inspecção, se determina se o sistema, ou os seus componentes, cumpre as condições impostas pelos requisitos. Muitas vezes a validação dos requisitos só é feita após o desenvolvimento, o que torna o processo bastante ineficiente.

3.7.1 Modelação de sistemas não triviais

De forma a se perceberem algumas das lacunas apresentadas pela linguagem no que respeita à aquisição, e correcta modelação, dos requisitos e funcionalidades do sistema que se pretende desenvolver, apresentam-se os passos mais importantes da modelação, de forma a aquilatar-se as limitações existentes.

Considere-se um sistema de gestão de uma biblioteca [dS02], que constitui um problema de complexidade controlada, mas que expressa perfeitamente as lacunas do processo de modelação. Considere-se que neste sistema existem dois tipos de utilizadores: os *Bibliotecários* e os *UtilizadoresNormais*. Cada um destes dois actores tem acesso a casos de uso diferenciados, sendo que existem claramente restrições ao conjunto de funcionalidade que um utilizador normal pode invocar. De forma contrária um bibliotecário, que é neste contexto o super-utilizador do sistema pode invocar todas as funcionalidades a que o utilizador normal tem acesso. Para facilitar a descrição do acesso a casos de uso por parte dos intervenientes no sistema, considere-se que existe um actor designado por UtilizadorBiblioteca que é obtido por generalização dos actores UtilizadorNormal e Bibliotecário. Um UtilizadorNormal é um actor que representa os utilizadores regulares de uma biblioteca, aqueles que requisitam livros e os devolvem quando o prazo termina. O Bibliotecário representa o operador do sistema e é responsável pela manutenção do mesmo.

A Figura 3.14 apresenta o diagrama de casos de uso para o sistema de software que corresponde à biblioteca.

Os casos de uso que se podem identificar são os usuais num sistema deste tipo e o actor UtilizadorNormal pode efectuar um série de casos de uso que determinam que possa "Procurar Livro", "Ver Estado de Livro", "Ver Livros Requisitados por Utilizador", "Consultar Livros", os quais correspondem ao comportamento expectável de um utilizador deste tipo.

Já o actor Bibliotecário, como é para todos os efeitos um super-utilizador, tem como casos de uso: "Gerir Info Livros", "Manutenção Utilizadores", "Recepção Livro", "Remover Livro", "Renovar Empréstimo", entre outros.

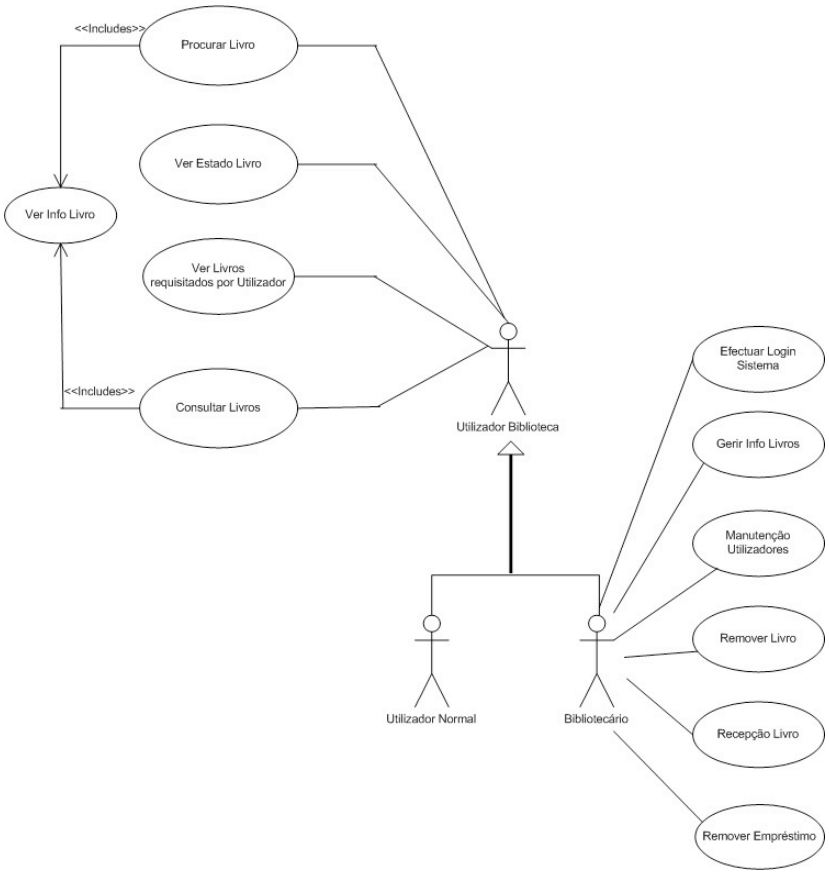


Figura 3.14: Diagrama de casos de uso da biblioteca.

O diagrama de classes de alto nível é apresentado na Figura 3.15. Neste diagrama é apresentado o diagrama abstracto que corresponde à definição do domínio do problema. Neste diagrama existem dois conceitos que importa referir e que correspondem

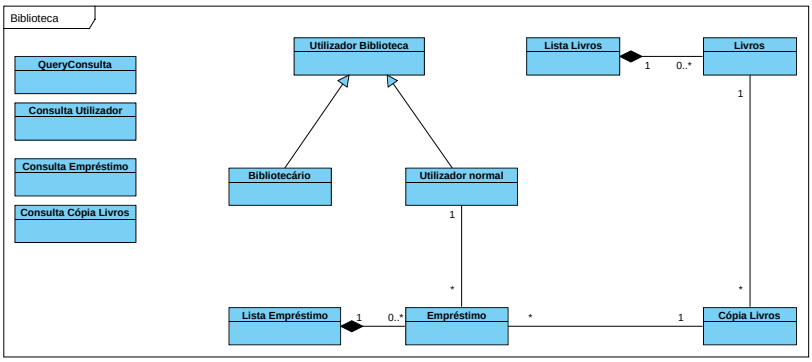


Figura 3.15: Excerto do diagrama de classes para o sistema de gestão da biblioteca.

a entidades diferentes no diagrama de classes. As classes que não contém nenhum estereótipo são aquelas que correspondem às entidades do sistema, como um utilizador as consegue percepcionar. Correspondem à concretização em conceitos do paradigma dos objectos das entidades que fazem parte do domínio do problema. As outras classes estão sinalizadas como possuindo o estereótipo << *control* >> [Jacobson 92], significando que são classes que fazem parte do comportamento do sistema software mas não correspondem a entidades perceptíveis do exterior do sistema. Estas classes são necessárias para a concretização da solução software, embora a sua descrição e relações, possam variar de solução para solução.

Na Figura 3.15 são apresentadas algumas destas classes, sendo possível que uma outra equipa de projecto chegue a definições arquitecturais diferentes da concretização do sistema, isto é, podem existir diferentes soluções para o mesmo domínio de negócio.

Descrição do comportamento

A modelação do comportamento de um sistema começa pelos casos de uso e a partir dele com o cenário que ele permite traçar, possibilita-se que se vá refinando a descrição do comportamento exibido pelo sistema. Seja o caso de uso "Procurar Livro"que permite a um utilizador normal consultar a base de livros disponíveis de forma a encontrar o livro desejado. O caso de uso "Procurar Livro", que não é especificado de forma alguma no diagrama de casos de uso atrás apresentado, e tendo em linha de conta, o que o autor sabe de como funciona uma biblioteca, é a concretização de uma funcionalidade que requer algumas pré-condições.

Tendo em conta a funcionalidade que existe num sistema de informação de uma biblioteca, é natural que o utilizador só possa procurar o livro que deseja se estiver ligado ao sistema. Isto é, antes de ser possível procurar o livro é necessário que o caso de uso que efectua a ligação ao sistema tenha sido previamente executado. Mesmo esta pré-condição não é suficientemente forte, na medida em que a execução desse caso de uso pode não ter resultado em sucesso.

É necessário que a execução de um caso de uso possa implicar que uma determinada ordem temporal tenha sido respeitada. É também necessário que a execução de um caso de uso esteja sujeita a regras básicas de verificação de condições booleanas associadas aos casos de uso.

Podemos enunciar alguns problemas de modelação:

Problema de modelação: Os casos de uso na sua forma original não

contêm informação sobre a precedência temporal em relação a outros casos de uso, nem prevêem a utilização de pré e pós-condições, como guardas da sua execução.

Este problema de modelação enuncia que a partir de um caso de uso, tomado de forma isolada, não é possível apenas recorrendo a ele, ter informação sobre a sua funcionalidade, nem é possível explicitar as condições que possibilitam a sua execução. As dependências temporais podem ser explicitadas recorrendo a outros diagramas, como sejam os diagramas de estado ou de actividade. Os diagramas de estado permitem ter uma vista sobre as entidades que são abrangidas pelo caso de uso e para cada uma delas detalhar os eventos que são desencadeados e as condições em que esses eventos podem ser executados. É assim possível utilizar essa informação para compor a descrição do caso de uso. O diagrama de actividade permite relacionar diversos actores do sistema e temporalmente definir as acções que são tomadas e a sequência a estabelecer entre elas. Estes diagramas apresentam uma vista parcial do sistema na medida em que focam apenas sobre uma determinada entidade ou acção, mas a combinação de todos estes diagramas dão uma visão completa do sistema. A definição de como é que se faz esta combinação é também uma melhoria que se introduz ao processo de modelação.

A própria definição dos diagramas de estado e de actividade coloca o problema de estes diagramas não serem contextualizados em relação ao seu estado inicial. Quando num diagrama de estados se identifica o estado inicial não é perceptível onde é que esse estado se localiza no fluxo de controlo do sistema. Esse problema não está resolvido na UML e para tal seria necessário que existissem novos construtores na linguagem. Uma solução possível para este problema passa por analisar as pré e pós-condições que se devem associar e por um processo de análise determinar os traços possíveis (as strings) da máquina de estados que pode ser criada.

Problema de modelação: Em UML não está definido o ponto de entrada no sistema, determinado pela invocação de um caso de uso.

Uma alternativa para suprir esta dificuldade seria recorrer a um conceito similar retirado da Interacção Humano-Computador (IHC) , utilizado na modelação de tarefas que é a Hierarchical Task Analysis (HTA) [Dix 04]. No entanto não existe um mapeamento linear entre os modelos de tarefas e as actividades, é possível utilizar estes para descrever o caso de uso em questão. Pode utilizar-se a informação recolhida nas tarefas resultantes de uma abordagem ao estilo da HTA para ter informação de como é

que o sistema hierarquiza e dispõe a execução das actividades que os seus casos de uso encerram. A utilização de diagramas de estado, ou actividade, conjuntamente com os casos de uso constitui-se como uma alternativa de incremento de semântica, visto que permite que os casos de uso definam a funcionalidade a nível macro e os diagramas de comportamento de estados ou actividade, especifiquem a um nível muito mais fino as interacções e trocas de mensagens.

Refinamento do fluxo de controlo

Grande parte das tarefas requer tanta informação sobre o domínio do problema como informação proveniente dos seus utilizadores. É necessário descrever quais são os objectos que em determinado momento estão a ser alvo de interacção, bem assim como os objectos que são criados e destruídos em cada modelação de uma actividade. A informação existente nos diagramas sobre o fluxo do controlo entre objectos especifica a forma como os conceitos do domínio do problema são utilizados na prossecução dos casos de uso desencadeados pelos utilizadores. No entanto, esta especificação do fluxo de controlo entre objectos, não permite descrever o comportamento dos objectos no que concerne às actividades que executam.

> **Problema de modelação**: A decomposição efectuada para especificar o comportamento dos objectos, origina diagramas de estado e de actividades complexos. A necessidade de modelar comportamentos concorrentes, independentes e repetidos é propícia a diagramas extensos com acrescida dificuldade de leitura e sujeitos a erros de interpretação.

A necessidade de, mesmo num sistema que, como este é de complexidade reduzida, especificar situações como a saída de um utilizador do sistema, a capacidade de este invocar acções em paralelo, ou a especificação de independência na ordem das acções conduz a que os diagramas fiquem muito complexos. Este problema de modelação não é reflexo da inexistência de suporte na linguagem a estas descrições. Pelo contrário, resulta do facto da linguagem oferecer construtores que devido à sua difícil capacidade de parametrização possam não propiciar que certos padrões de comportamento sejam abstraídos e reutilizados noutros contextos[8]. Imagine-se a simples tarefa de requisitar um livro e o facto de que o utilizador poder estar ligado no sistema, como convidado, não tendo previamente fornecido a sua identificação. Quando procura um

[8]Note-se que alguns dos alterações induzidas na versão 2.0 da UML procuram dotar a linguagem desta capacidade.

livro e o quiser requisitar terá de efectuar o registo no sistema, enquanto que se já estiver registado, não precisa de efectuar este passo e a sua informação de credenciais é automaticamente alimentada. Este pequeno exemplo de comportamento, torna os diagramas comportamentais complexos em demasia e desnecessariamente algorítmicos.

A especificação de actividades que podem ocorrer em paralelo pode ser obtida através da combinação de transições, forks e joins, enquanto que a utilização de ramos diferentes do mesmo diagrama de actividades pode ser utilizada para a modelação da escolha. O diagrama de actividades, por si, não é suficiente para descrever o comportamento na medida em que não consegue capturar correctamente as acções que decorrem no contexto de um estado de uma entidade. Acresce a dificuldade que o diagrama de estados proporciona uma visão da entidade e não favorece o raciocínio sobre o que é que se passa a nível dos objectos do sistema.

O diagrama de sequência apresenta problemas no que concerne à descrição de comportamentos que implicam a existência de precedências temporais entre as acções existentes no diagrama.

> **Problema de modelação**: Os diagramas de sequência não permitem descrever, de modo facilitado todas as dependências temporais entre acções, quando elas são complexas ou não são evidentes.

Embora seja possível especificar este tipo de conhecimento, exige no entanto, que os diagramas sejam complexos, na medida em que não é simples a passagem do diagrama de estado e/ou actividades, possivelmente com vários fios de execução, para o diagrama de sequência, em que a única possibilidade que existe é a de representar sequencialidade temporal e também detalhar algumas restrições temporais no que concerne aos tempos de execução das acções. A introdução de *frames* de interacção nos diagramas de sequência vem tornar mais modular a descrição destas necessidades, mas não torna o diagrama de mais fácil leitura. Num sistema complexo, torna-se por vezes mais vantajoso criar um diagrama de sequência para cada fio previsível de interacção do que criar um único diagrama de sequência. Em UML 2.0 os *novos* diagramas de interacção procuram colmatar este excesso de complexidade conjugando diagramas de actividade e de estado num único diagrama.

O exemplo do sistema de gestão da biblioteca, permite estabelecer que é possível modelar em UML sistemas de comportamento complexo, mas também demonstra que o resultado da modelação consiste em diagramas complexos nos quais o detalhe é muito elevado e de difícil leitura e verificação de correcção. Este exemplo possibilitou identificar as componentes da UML que estão mais desguarnecidas no que respeita às

capacidades de modelação e onde faz sentido investir na construção de um nível de reforço da semântica dos diagramas. Em resumo as dificuldades encontradas, podem sintetizar-se nos seguintes pontos:

1. A modelação de funcionalidades complexas com o recurso aos diagramas de casos de uso e à descrição textual de cada um deles;

2. A identificação de pontos de entrada nos casos de uso da aplicação, de forma a relacioná-los com os diagramas de estado e de actividades. Note-se que este é um problema que não ocorre em aplicações orientadas ao fluxo de dados onde é possível determinar os pontos de entrada nos casos de uso;

3. A descrição de diagramas de actividades e diagramas de estado muito complexos, sendo necessário representar interacções complexas e com requisitos de concorrência e alternativa, torna os modelos difíceis de interpretar e sujeitos a erros,

4. A utilização de diagramas de sequência é mais simples e directa se as dependências temporais puderem ser separadas e tratadas isoladamente.

As dificuldades de modelação identificadas permitem aquilatar da dimensão dos esforços necessários para descrever correctamente os diagramas necessários. Apesar de se terem identificado dificuldades existem no entanto algumas conclusões que é possível retirar deste exercício:

1. A incapacidade do diagrama de casos de uso e da descrição textual dos casos de uso, na sua forma mais simples, para apreender a informação necessária da fase de análise.

2. A necessidade de combinar vários diagramas numa ordem que é possível de ser enunciada, sendo que o processo de modelação é definitivamente iterativo e não existe uma separação rígida entre as diversas vistas.

A Tabela 3.5 apresenta para cada fase da tarefa de modelação as ferramentas, isto é os diagramas, que devem ser utilizadas.

Esta tabela apresenta em súmula da metodologia proposta e que se aborda em detalhe em capítulo posterior. Existem aspectos a melhorar, ou mesmo a adicionar, na perspectiva de incremento do poder expressivo da linguagem de forma a ser mais conveniente a sua utilização nas diversas fases do processo de desenvolvimento de um sistema software complexo e de larga escala.

Tarefa	**Diagrama UML**
Captura de Requisitos	diagrama de casos de uso + descrição textual de casos de uso
Domínio da Aplicação	diagrama de classes
Identificação das Funcionalidades	diagrama de casos de uso + diagrama de estados + diagrama de actividades + diagramas de sequência

Tabela 3.5: Sumário da utilização usual dos diagramas.

As dificuldades de modelação identificadas atrás surgem quando se pretende modelar sistemas que apresentam um comportamento que envolve interacções complexas, típicas de sistemas concorrentes, distribuídos, tempo-real ou apenas interactivos. Note-se que grande parte dos requisitos de modelação necessários encontram-se nos sistemas interactivos, que são eminentemente complexos, são reactivos, e partilham muitas das características dos outros sistemas.

A avaliação que é possível fazer da UML indica que a notação cobre de forma muito positiva a descrição estrutural das entidades e dos seus relacionamentos, isto é a componente arquitectural, mas não aborda da forma mais conveniente a descrição comportamental, isto é, a forma como os utilizadores interagem com o sistema e como esta interacção afecta o estado global do mesmo. Esta menor capacidade de modelação destes aspectos não pode ser escondida nem contornada, visto que grande parte dos sistemas software actualmente em desenvolvimento, ou em manutenção, exibem comportamentos complexos.

3.7.2 Modelação de diferentes tipos de sistemas software

Sistemas Concorrentes

Num sistema software com concorrência interna existem tipicamente várias actividades que ocorrem em paralelo. Estes sistemas são intrinsecamente distintos dos sistemas sequenciais, em que as actividades são executadas em estrita sequência. Nos sistemas concorrentes a ordem dos eventos que ocorrem pode não ser previsível e frequentemente os eventos de diversos fios de execução ocorrem simultaneamente.

Numa aplicação deste tipo encontram-se normalmente vários objectos activos [Gomaa 00], cada um controlando um fio de execução. Além da troca de mensagens síncronas, a troca de mensagens assíncronas é também suportada, pelo que um objecto pode enviar

uma mensagem e continuar a sua execução, sem ficar à espera da resposta. Se o objecto destino estiver ocupado na sua execução, a mensagem é colocada em fila de espera até que possa ser atendida[9].

O conceito de processo concorrente é central no desenvolvimento das aplicações. Podemos encontrar processos concorrentes em aplicações como sejam sistemas interactivos para um domínio aplicacional específico, sistemas operativos, aplicações Web interactivas, sistemas de base de dados, entre outras.

O processo de desenvolvimento de aplicações concorrentes necessita que existam mecanismos com a capacidade de capturar e estruturar um modelo do sistema que detalhe todas as suas particularidades e características intrínsecas. A captura dos requisitos, a descrição do comportamento associado aos requisitos funcionais, a troca de mensagens entre os objectos, são aspectos que são necessários registar detalhadamente no modelo.

A Tabela 3.6 sumaria os necessidades de modelação em UML das aplicações concorrentes.

Necessidade	**Diagrama UML**
Captura de Requisitos	diagrama de casos de uso + descrição textual dos casos de uso
Domínio da Aplicação	diagrama de classes
Modelo Concreto	diagrama de classes + identificação objectos activos + *patterns* comportamentais de concorrência
Descrição das Funcionalidades	diagrama de casos de uso + diagrama de estados + diagrama de actividades + diagramas de sequência

Tabela 3.6: Diagramas utilizados para a modelação de sistemas concorrentes.

Sistemas Distribuídos

Um sistema software distribuído é uma aplicação concorrente que executa num ambiente que é disperso geograficamente, sendo composto por um conjunto de nós. Estes nós estão interligados entre si através de uma rede e cada um dos nós pode fornecer um conjunto determinado de serviços que são requisitados por outros nós.

[9]Embora este comportamento possa ser diverso consoante a implementação do contexto que suporta esta fila de espera.

Um factor de complexidade existente nos sistemas distribuídos deriva do facto de ser necessário modelar os impactos que a indisponibilidade de um nó, quer por falha do próprio ou da rede que o interliga aos demais, provoca no restante sistema. Como exemplos de aplicações distribuídas podem ser referidas as bases de dados distribuídas, sistemas de ficheiros distribuídos, entre outros.

A problemática da falha de um nó, ou da rede, é algo que se deve ter em conta na altura da análise e subsequente modelação. É necessário criar as entidades que sinalizam erros (excepções) e classificar as entidades do sistema, através da inclusão de classificadores (estereótipos) de forma a permitir separarem-se domínios distintos de acordo com a tipologia do sistema.

A Tabela 3.7 apresenta as ferramentas da linguagem para a modelação de sistemas distribuídos.

Necessidade	**Diagrama UML**
Captura de Requisitos	diagrama de casos de uso + descrição textual casos de uso
Domínio da Aplicação	diagrama de classes
Identificação das Funcionalidades	diagrama de casos de uso + diagrama de estados + diagrama de actividades + diagramas de sequência
Modelo Concreto	diagrama de classes + estereótipos de classificação + entidades de excepção
Informação de instalação	diagrama de deployment

Tabela 3.7: Diagramas utilizados para a modelação de sistemas distribuídos.

Sistemas de Tempo Real

Um sistema de tempo real pode ser descrito como sendo um sistema concorrente com restrições temporais. Os sistemas de tempo real são, na maioria das vezes, complexos devido a terem de lidar com múltiplas fontes independentes de eventos. Esses eventos têm tempos de chegada que não são previsíveis e por vezes tem restrições temporais associadas ao fornecimento da resposta.

Os aspectos relativos à integração com outros sistemas, vistos como sendo caixas negras (sejam hardware ou software), bem como as restrições existentes, a nível temporal e de interligação de componentes, devem estar presentes no modelo, de forma a ser possível capturar a informação necessária para a efectiva descrição dos sistemas.

As aplicações de tempo real podem beneficiar na linguagem UML de uma extensão própria, na medida em que a OMG definiu oficialmente um perfil, para a modelação deste tipo de sistemas, designado de *Schedulability, Performance and Time Profile*.

A Tabela 3.8 apresenta as ferramentas da linguagem para a modelação de sistemas tempo real. Note-se que tendo em conta que a natureza destes sistemas é de considerar que as restrições temporais têm de estar expressas nos modelos, sendo que é possível adicionar aos diagramas de sequência informação sobre as restrições temporais. É no entanto, muito complexa a modelação deste tipo de sistemas, devido às dificuldades da UML na criação de diagramas simples para sistemas onde é necessário especificar dependências temporais, escolhas e ciclos.

Necessidade	**Diagrama UML**
Captura de Requisitos	diagrama de casos de uso + descrição textual dos casos de uso
Domínio da Aplicação	diagrama de classes
Identificação das Funcionalidades	diagrama de casos de uso + diagrama de estados + diagrama de actividades + diagramas de sequência com estampas temporais e frames de interacção

Tabela 3.8: Diagramas utilizados para a modelação de sistemas de tempo real.

Sistemas Interactivos

Um sistema interactivo designa o sistema computacional que prevê a existência de comunicação com o utilizador. Além da necessidade de modelar a camada computacional do sistema é necessário também descrever a parte do sistema software que especifica a componente da interacção com o utilizador. A modelação desta camada é especialmente complexa, na medida em que a estruturação do diálogo com o utilizador é tipicamente complexa.

As três fases da construção da camada interactiva que interessa prever são a Análise de Requisitos, a Análise de Tarefas e o Desenho do Diálogo. Para cada uma destas actividades, por esta ordem, pode ser proposto um método de abordagem e utilização da UML, de forma a capturar a informação pretendida [Campos 06].

Faz sentido detalhar um pouco as preocupações que a modelação deste tipo de sistemas introduz, visto que os sistemas interactivos possuem, em boa medida, grande parte da complexidade dos sistemas anteriormente referenciados.

O processo pode ser descrito de forma sucinta, nos seguintes passos:

i) Análise de requisitos — utilizam-se os diagramas de casos de uso para capturar os requisitos expressos pelos utilizadores do sistema. Este é o diagrama mais adequado para esta fase até pela definição de caso de uso, que diz tratar-se de uma descrição informal dos requisitos funcionais, dos actores envolvidos e dos resultados produzidos. Para melhorar a capacidade de expressividade da descrição associamos a cada um dos casos de uso a descrição de um ou mais cenários, de forma a capturar a este nível toda a informação que se pode obter e que é importante para a camada interactiva.

Requisitos de usabilidade, tal como outros requisitos não funcionais, podem ser incluídos, nesta fase, como notas ou restrições ao modelo.

ii) Análise de tarefas — utiliza-se a informação obtida anteriormente para extrair a informação necessária à especificação das tarefas. Abordagens como as CTT [Paternò 00, Paternò 99] capturam formalmente esta descrição, mas em UML podemos recorrer aos diagramas de actividade de forma a garantir um nível de expressividade semelhante. Devido ao facto de a modelação de este tipo de sistemas não ser uma preocupação nativa da UML, é por vezes necessário tornar mais os diagramas de actividade produzidos mais complexos do que o ideal.

Aos diagramas de actividade foi acrescentada a capacidade de delinear regiões interrompíveis (no que concerne ao controlo de fluxo), o que permite a especificação de operações de cancelamento da tarefa (e.g. *Cancel*). Falta ainda em relação à expressividade oferecida pelo CTT, a capacidade de certos diálogos poderem ser temporariamente suspensos e posteriormente retomados (operador `|>`). Não é também linear a capacidade de efectuar estruturação e encapsulamento explícito de diagramas de actividade, pelo que a descrição da análise de tarefas obtida necessita por vezes de documentação de apoio.

iii) Desenho do diálogo — nesta fase, através da informação existente nos diagramas de casos de uso e de actividade, identificam-se os momentos de interacção necessários à aplicação (ecráns e *forms*) e, em função das descrições existentes nos diagramas de actividade desenvolvem-se diagramas de transição de estados. Os diagramas de estados assim obtidos, não apresentam, a nível de fluxo de informação, alterações significativas em relação ao correspondente diagrama de actividades. No entanto, o foco de atenção passa das actividades realizadas por cada actor, para o impacto que essas actividades têm na componente interactiva.

Ao descrever as transições torna-se também possível decorá-las com os métodos que a camada de negócio deverá disponibilizar para suportar o comportamento descrito.

Este passo metodológico permite que se vá refinando o modelo e que sejam levantados os métodos da camada de negócio que implementarão as acções associadas à camada de apresentação. Sendo este processo nitidamente iterativo, através da descrição dos diagramas de transição de estado vai-se completando a informação sobre o sistema e, além de se especificar a componente relativa ao controlo de diálogo, também se vai adquirindo informação sobre a arquitectura da camada de apresentação.

Descritas as fases da modelação da interface e sua correspondente incorporação na linguagem UML, importa referir algumas pré-condições que regem a utilização da linguagem:

- existe um diagrama de actividade para cada caso de uso descoberto na análise de requisitos (cada diagrama de actividades deve conter toda a informação dos possíveis cenários que se podem obter a partir de um caso de uso);
- deve existir um caminho no diagrama de estado para cada linha (fluxo) que se possa traçar no diagrama de actividades, e
- o conjunto dos métodos obtidos na decoração dos diagramas de transição de estados para uma determinada entidade representa o comportamento exibido por essa entidade (logo, não existe comportamento numa entidade que não seja obtido a partir do processo de construção do diagrama de transição de estado).

Estas condições, não são apenas respeitantes ao processo a adoptar para a modelação de sistemas interactivos, mas podem ser aplicadas aos tipos de sistema anteriormente apresentados. Como se disse os sistemas interactivos são por definição sistemas software complexos e nos quais os requisitos de concorrência, tempo e até mesmo distribuição, se podem fazer sentir.

A Tabela 3.9 faz o sumário das ferramentas utilizadas para cada uma das necessidades no processo de desenvolvimento de sistemas interactivos.

3.8 Abordagens Alternativas

Com o objectivo comum de enriquecer a UML com uma semântica bem definida e precisa, várias abordagens distintas têm vindo a ser propostas pela comunidade. Essas abordagens podem ser caracterizadas em três grandes famílias:

1. uma abordagem de construção de métodos e linguagens formais;

Necessidade	**Diagrama UML**
Captura de Requisitos	diagrama de casos de uso + descrição textual dos casos de uso
Domínio da Aplicação	diagrama de classes
Análise de Tarefas	diagrama de actividades
Desenho do Diálogo	diagrama de casos de uso + diagrama de estados + diagrama de actividades + diagramas de sequência
Modelo Concreto	diagrama de classes + *patterns* de interacção

Tabela 3.9: Diagramas utilizados para modelação de sistemas interactivos.

2. uma abordagem orientada à fundamentação e raciocínio ao nível do meta-modelo UML, e

3. uma abordagem em que são criados novos construtores e diagramas.

Cada uma destas abordagens tem como objectivo a construção de uma camada semântica a adicionar à linguagem, de modo a obviar alguns dos problemas atrás identificados. Por fim, registe-se a existência de trabalhos que tem por base a validação e verificação de requisitos e que tocam as abordagens atrás referenciadas, com o intuito de promover de forma eficaz e concreta o raciocínio e operacionalização da fase de análise.

3.8.1 Abordagem Formal

Em [Grieskamp 00] é utilizada a notação formal Z [Spivey 89] para fornecer uma representação do modelo dos casos de uso de forma a aplicá-la a vários exemplos de casos de uso. Parte-se da definição do que é um caso de uso, nomeadamente de que os factos observáveis são sequências de interacções, sendo que tal constitui um diálogo. Um caso de uso é assim um fragmento de um diálogo, isto é, uma sub-sequência do diálogo total. Os casos de uso identificados não são descritos em diagramas de casos de uso, tal como se faz em UML, mas são descritos em Z, permitindo acrescentar a capacidade de efectuar testes de consistência do modelo obtido. Existe um foco na operacionalização ao permitir a utilização de Z para testar os modelos criados. É possível dado um conjunto de casos de uso escritos em Z, informação relativa a um caso de teste (dados de entrada) e informação relativa aos resultados esperados (dados de saída), saber se a

execução dos casos de uso é correcta, isto é, se a formalização dos requisitos foi feita com sucesso. Neste trabalho endereçam-se problemas decorrentes da introdução de concorrência nos casos de uso e as sequências de acções são agregadas em diálogos, permitindo depois efectuar testes sobre os traços do diálogo.

Esta abordagem não utiliza a linguagem UML, visto que depois da identificação dos casos de uso, passa-se directamente para a escrita em Z, permitindo que as regras definidas informalmente relativamente aos requisitos sejam transformadas em mecanismos de maior rigor e de maior abstracção. No entanto, a distância entre os casos de uso especificados desta forma e o conhecimento que os utilizadores têm da linguagem dificulta a aplicação directa desta abordagem.

Em [Moreira 99] é abordada uma estratégia semelhante com o uso de Object-Z, na medida em que a partir dos diagramas de Caso de Uso são gerados modelos na linguagem formal e é sobre esses modelos que o raciocínio e execução é efectuado. Os casos de uso são posteriormente representados por diagramas de sequência, permitindo explicitar a ordem temporal das trocas de mensagens entre os objectos participantes. O trabalho reconhece que o processo de formalização não é imediato e de aplicação instantânea, visto ser necessária a existência de familiaridade com a linguagem formal, sendo preferível, e desejável, a existência de um processo automatizado de passagem da descrição do caso de uso (informal) para uma descrição formal.

O processo de criação dos casos de uso está representado na Figura 3.16.

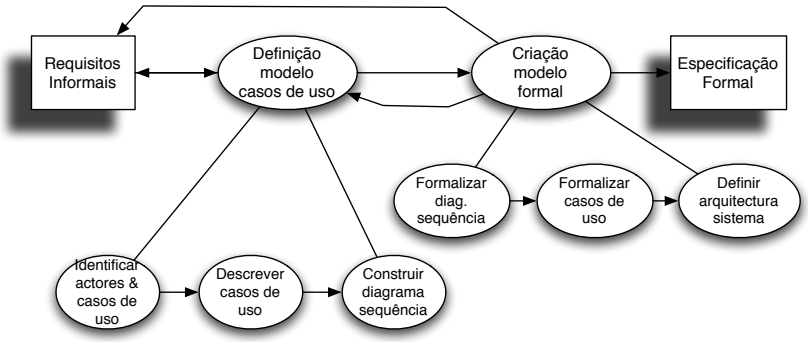


Figura 3.16: Processo de formalização dos casos de uso.

A arquitectura do sistema, que se obtém corresponde a um arquétipo formal que se pode utilizar para início do processo de construção do diagrama de classes completo. O processo é iterativo e incremental, não sendo suposto que os casos de uso estejam todos detalhados, e identificados, antes de começar o processo de formalização. O processo é nesse sentido evolutivo, na medida em que se permite que se comece com

os casos de uso conhecidos, se proceda à sua formalização e com a realimentação dada pela validação, com o utilizador, se vá refinando o modelo. Apesar deste pressuposto, não é claro neste trabalho como é que se faz a junção, o *somatório*, de todas as vistas parciais que vão sendo obtidas à medida que se analisa novo caso de uso. Este trabalho apresenta como mais-valia, o facto de

i) encorajar desde o início a utilização de métodos, e linguagens, formais, o que beneficia a qualidade do sistema final;

ii normalizar as diversas notações que podem estar num diagrama de casos de uso numa única notação formal, precisa e rigorosa;

iii permitir elaborar sobre os traços que é possível criar a partir dos diagramas de sequência, e

iv permitir raciocinar sobre o sistema, na medida que a linguagem em que está especificado tem uma semântica matemática (bem fundada).

No entanto, o facto de se criar um outro nível de representação do sistema, em Object-Z, não promove *per si* a qualidade final do sistema, a não ser que a transformação da linguagem formal para código numa linguagem de programação possa ser automatizado.

Ainda nos trabalhos que criaram uma representação numa linguagem formal, uma modelação completa em termos de utilização do Z foi feita por [France 00], na qual são utilizados os diagramas de classe e de casos de uso e é feita a sua transformação em especificações em Z. A análise dos modelos gerados em Z e a própria transformação dos modelos permite detectar as inconsistências e ambiguidades na modelação feita a partir dos requisitos fornecidos.

No trabalho de [Eshuis 02], a mesma raiz do problema é abordada de forma semelhante, sendo que o foco é dado aos diagramas de actividade que podem ser criados a partir dos diagramas de casos de uso. Neste trabalho a contribuição é muito mais focada na componente de *workflow* que é de difícil representação ao nível dos casos de uso. É utilizado *model checking* para a verificação de requisitos funcionais em especificações de *workflow*. A temática do trabalho reside na definição de uma semântica formal para os diagramas de actividade que são escritos em complemento aos de caso de uso para descrever as tarefas de *workflow*, sendo que questões como paralelismo e concorrência são naturalmente endereçadas, não conseguindo estas ser expressas a nível dos diagramas de casos de uso.

A problemática da consistência de informação entre os vários níveis de diagramas que se podem retirar dos diagramas de casos de uso é abordada também pelos trabalhos de [Paige 02] e [Övergaard 98]. Em [Paige 02] a preocupação dominante é a resultante das possíveis inconsistências existentes entre as diversas vistas de um mesmo modelo. Como em UML as vistas podem ser construídas de forma independente pode ser verificado se são de facto coerentes, e completas. Apenas garantidas estas premissas se pode pensar em passar dos modelos à geração de código. Essa preocupação é também evidente no trabalho de [Straeten 04] na preocupação que expressa na consistência do modelo e na parte comportamental. O facto de não existir uma interpretação geralmente aceite como correcta da componente comportamental dos diagramas UML, introduz um grau de incerteza na possibilidade de incorporação de transformações dos modelos. Os diagramas de estados e de sequência criados com base num caso de uso devem ser consistentes no que respeita à herança de comportamento.

Nesta dissertação a consistência do modelo pretende ser assegurada desde a escrita de diagramas de caso de uso recorrendo a sobre-especificação dos mesmos e a complementaridade com diagramas de comportamento. Os trabalhos de [Paige 02] e [Straeten 04] são a esse nível importantes referências para validar a consistência do modelo.

3.8.2 Fundamentação do Meta-modelo

Com o objecto de melhorar a semântica da UML e fornecer-lhe uma que seja rigorosa e precisa, existem inúmeros trabalhos cuja incidência se faz sentir na definição do meta-modelo. Isto deve-se a que o meta-modelo UML (versões 1.x) não é completamente bem definido, no sentido formal da expressão, sendo composto por descrições feitas em linguagem natural. Daí existir um espaço amplo de intervenção em que se pode melhorar a semântica dos modelos, pela melhor definição dos construtores da linguagem. Ou seja, antes de se criarem novos construtores para introduzir novos conceitos, ou simplesmente melhorar os anteriores, pode ser feito um trabalho de reorganização, que permita estabelecer a semântica do que actualmente existe. Estas acções permitem que a interpretação dos diagramas passe a ser única, o que é um resultado muito importante.

No que se refere à fundamentação a nível do meta-modelo do UML podem ser citados os trabalhos de [Wegmann 00] em que de forma consistente aborda a possibilidade de conjugar vários diagramas de forma a acrescentar expressividade e diminuir o risco de a modelação ficar afectada pelas insuficiências da notação. Refira-se também o tra-

balho em profundidade feito na resposta à especificação do UML 2.0, nomeadamente em relação ao OCL que pode ser encontrada em [Consortium 02] e em documentação subjacente do mesmo grupo de trabalho [Maskeri 02].

3.8.3 Extensão a construtores e perfis

Na vertente em que se acrescentam novos construtores diagramáticos e se altera o processo e a linguagem, existem vários trabalhos que abordam a problemática desta forma. Entre eles podem-se identificar os trabalhos da comunidade multi-agente [Bauer 01, Odell 00] que deram origem a uma linha de acção na especificação da versão 2 do UML, no âmbito da FIPA. As alterações introduzidas pela *agent UML*, são feitas ao nível das alterações aos diagramas, de forma a simplificarem a modelação de comportamentos complexos. Como foi apresentado em secção anterior os diagramas de comportamento em UML podem ser de difícil construção, e leitura, quando o sistema a modelar apresenta um comportamento - diálogo entre as entidades - complexo. Acresce o facto de nos sistemas multi-agente ser frequente a criação de novos actores, novos agentes, não sendo simples a nível dos diagramas de objectos e de interacção a sua representação em compreensão. Daí a comunidade multi-agente ter introduzido um conjunto de novos construtores que permitem simplificar a construção dos diagramas, permitindo abstrair complexidade e juntar no mesmo diagrama notações de vistas diferentes da UML. O resultado final é uma simplificação da escrita dos diagramas obtidos a partir dos diagramas de caso de uso e que especificam o comportamento desejado.

Ainda nesta abordagem surgem extensões aos diagramas de casos de uso por incorporação de outros conceitos semelhantes [Amyot 00]. Procura-se diminuir a distância semântica entre os requisitos e os diagramas de casos de uso a eles associados, da restante modelação recorrendo a diagramas de comportamento. O trabalho de Amyot tem por base o facto de que os sistemas reactivos dependem fortemente dos aspectos comportamentais, descritos muitas vezes como cenários de um caso de uso. A utilização de Use Case Maps permite descrever relações causais entre as entidades. Providencia-se desta forma uma representação visual dos casos de uso em termos de causalidade, responsabilidades e consequências das acções.

Entre os trabalhos mais próximos do desta dissertação identificam-se alguns que abordaram o problema com a mesma intenção e que optaram por soluções semelhantes. Alguns desses trabalhos abordam o problema para resolverem questões de uma área específica, os sistemas interactivos, onde a ambiguidade de alguns diagramas de casos de uso se fazem sentir, por serem manifestamente pouco expressivos. Os sis-

temas interactivos são inclusive um bom caso de estudo na medida em que o nível de expressividade exigido é muito grande e a mesma modelação pode ser derivada de sistemas software distintos. Acresce a esta necessidade o facto de a prototipagem nos sistemas interactivos ser muito importante na validação dos requisitos. A esse nível a combinação de diagramas é preconizada em [Campos 06], de forma a aumentar a capacidade de descrição de diálogos complexos.

O trabalho de [Nunes 00, Nunes 01] parte dos mesmos princípios que esta dissertação mas foca-se na área dos sistemas interactivos e nomeadamente na descrição da interface, ou seja, identifica a falta de suporte para a descrição destes sistemas na linguagem UML e no modelo de processo associado, propondo uma perfil para suprir estas lacunas. Esta falha é contornada pela incorporação de extensões baseadas num método de engenharia de software, o Wisdom. As extensões propostas permitem expressar a modelação de tarefas e possibilitam a descrição de padrões de interacção. Segundo o que é defendido, os modelos utilizados para suportar a concepção da aplicação não são adequados para garantir a concepção da interacção, tendo em conta o comportamento potencialmente complexo desta e a usabilidade que se quer garantir. A adopção de casos de uso reforça a importância da identificação dos requisitos e os diferentes actores que interagem com a aplicação, mas não proporcionam uma alternativa eficaz para a representação dos aspectos de usabilidade e interacção. Neste trabalho é proposta a utilização de diagramas de estados (e actividades) para detalhar os casos de uso.

Trabalho com motivação semelhante é apresentado em [dS00, dS02], ainda na área dos sistemas interactivos, em que se desenvolveu uma ferramenta - ARGOi - que importa modelos definidos em ArgoUML (que é baseada na UML padrão). Neste trabalho faz-se uma descrição exaustiva das lacunas da UML para a descrição de sistemas de software complexos, como sejam os sistemas interactivos, e incorporam-se técnicas baseadas em modelos de descrição de interfaces com o utilizador. Consegue-se assim uma ligação mais eficaz entre a interface com o utilizador com o sistema software em que ela assenta. O UMLi providencia uma notação diagramática aumentada para modelar interfaces com o utilizador e acrescenta também notação adicional aos diagramas de actividades de forma a descreverem comportamentos específicos dos sistemas interactivos. O trabalho fornece uma definição semântica bem definida com recurso ao uso da OCL e a ferramenta resultante é bastante expressiva.

Um trabalho com objectivos finais semelhantes é o apresentado em [Duran 04, Bernardez 04] no qual o resultado da modelação recorrendo a diagramas de casos de uso é analisado e considerado insuficiente. São estudados os defeitos e erros típicos na utilização de diagramas de casos de uso e o facto de a capacidade de expressividade

actuais destes influírem na qualidade do resultado obtido. Este trabalho foca-se no processo de descoberta de requisitos de forma a aplicar heurísticas no processo, de maneira a aumentar a expressividade dos diagramas de casos de uso obtidos.

O trabalho de [Goñi 04], é relevante pelo esforço que representa a construção de especificações UML precisas e não ambíguas em problemas típicos de concorrência. Este trabalho faz uso da OCL para provar que é possível com o recurso a uma linguagem de restrições preservar construtores típicos de concorrência e que é possível acrescentar aos modelos UML esta informação, combatendo a possível ambiguidade dos mesmos.

3.8.4 Validação e verificação

Uma área em que tem vindo a ser desenvolvido trabalho nas tarefas associadas à captura de requisitos e posterior validação é a que tenta acrescentar capacidade de formalização aos casos de uso, de maneira a que seja possível a automatização das tarefas de teste. Existem diversas abordagens que estudam a temática da validação por teste e da geração desses mesmos testes a partir de especificações formais. Exemplos dessa abordagem encontram-se em [Helke 97, Legeard 02, Tahat 01]. Estas iniciativas apresentam do ponto de vista da adequação à utilização por parte dos analistas algumas dificuldades devido ao uso de linguagens e métodos formais. O ciclo de vida da manutenção dos requisitos quando expressos em linguagens formais obriga a que tanto o analista como os utilizadores tenham conhecimentos para conseguir descrever os requisitos, bem como consigam validar e interpretar os resultados dos testes efectuados sobre os requisitos.

Existem outras abordagens, menos formais, que exploram a geração de cenários de teste e que que se baseiam no modelo dos casos de uso [Briand 02, Frolhich 00, Ryser 99]. Em [Frolhich 00] é descrita uma abordagem para gerar os cenários de teste a partir da descrição dos casos de uso com pré e pós-condições como defendido em [Cockburn 01]. Cada caso de uso é transformado numa máquina de estados e a verificação do teste faz-se com base nesse formalismo.

Uma abordagem baseada nas dependências entre os casos de uso, permitindo representar essas relações com recurso a notações visuais é apresentada em [Briand 02, Ryser 99]. Em [Ryser 99] o trabalho apresenta uma visão orientada aos cenários de forma a derivar casos de teste para validação de requisitos. Os cenários são formalizados com recurso a diagramas de estado e um grafo de dependências modela as relações entre os cenários. O trabalho apresentado em [Ryser 00] também utiliza um grafo de dependências, mas permite o seu refinamento em diversos tipos de dependência, como

sejam de abstracção, temporais e de recursos de dados. Estes trabalhos introduzem uma nova notação para complementar o modelo dos casos de uso, o que as tornam não padrão (standard) em termos do seu suporte por ferramentas.

Na mesma linha de investigação encontra-se o trabalho de [Briand 02] onde se propõe a utilização de um diagrama de actividades alterado para incluir as restrições dos casos de uso. Do diagrama de actividades, através da transformação num grafo pesado, são extraídas as expressões, que correspondem às sequências de casos de uso. Os cenários de teste podem ser obtidos de entre as sequências obtidas do grafo. Os diagramas de actividade são mecanismos que apresentam como principal vantagem o facto de serem facilmente compreendidos e terem a capacidade de expressar fluxo de controlo e as restrições ao mesmo. A introdução das *"swim lanes"*, nos diagramas de actividade permite associar ainda responsabilidades (que actores?, que papéis?) a cada uma das actividades em curso.

3.9 Resumo

Este capítulo abordou as notações de modelação e a sua importância no processo de desenvolvimento de software. A UML como linguagem de modelação foi apresentada e a sua adequação às metodologias orientadas aos objectos foi justificada. As características mais importantes da linguagem foram enunciadas e o seu carácter de notação comum para as actividades de modelação foi justificado e o conjunto de diagramas disponibilizados para o efeito foram sinteticamente descritos.

Tendo em atenção a temática deste trabalho, alguns dos diagramas da UML são importantes, nomeadamente os diagramas de casos de uso, de classe, actividade, estado e sequência. Daí, neste capítulo, terem sido brevemente apresentados com algum detalhe estes diagramas, enunciando as suas características mais importantes, de modo a que se compreenda a sua capacidade de modelação.

Apresentaram-se as alterações e modificações introduzidas pela versão 2.0 da UML. Essas alterações foram justificadas como tendo sido motivadas pela necessidade de aumentar o nível de rigor e precisão na definição da semântica da linguagem. As melhorias introduzidas permitem ter uma arquitectura da linguagem melhorada e com capacidade de automatizar a passagem dos modelos para código final.

Finalmente, fez-se uma referência à linguagem de escrita de restrições, OCL, para conhecimento sumário da mesma e avaliação da sua importância.

A necessidade de integração de técnicas formais na linguagem foi apresentada e

justificada como estratégia para a melhoria da qualidade dos modelos produzidos. A importância da descrição dos casos de uso e o papel que os utilizadores nele desempenham foi também abordado, bem como quais são os factores importantes a ter em conta nesta fase. Apresentou-se também um exemplo que permitiu identificar os principais problemas associados à modelação de sistemas. Foram enunciadas essas dificuldades, que resultam na maioria das vezes de descrições comportamentais complexas, e apresentaram-se formas de obviar essas lacunas de modelação. A tipologia dos diagramas UML necessários para descrever alguns tipos de sistemas foi fornecida, e a forma como abordar cada uma deles foi apresentada.

Por fim, enunciaram-se alguns trabalhos na área que ajudam a perceber o espaço de intervenção e a tipificar as diversas alternativas que têm surgido.

Capítulo 4

Uma Proposta de Processo de Modelação

4.1 Introdução

Neste capítulo apresenta-se e justifica-se o processo de integração de técnicas rigorosas na descrição de casos de uso, possibilitando dessa forma uma filosofia de construção dos restantes diagramas comportamentais do modelo. Tendo apresentado, em capítulos anteriores, a capacidade que a UML oferece para a descrição e modelação de sistemas de software e tendo também abordado os constrangimentos colocados quando a complexidade aumenta, neste capítulo apresenta-se um método de construção do modelo, que permite acrescentar informação àquela que é capturada nas metodologias tradicionais e que induz um acréscimo de qualidade às fases de análise e especificação. O capítulo anterior permitiu identificar as lacunas existentes nas várias ferramentas que a UML disponibiliza: a pouca carga semântica que é colocada nos diagramas e descrição dos Casos de Uso e que distorce o nível de rigor entre os diversos diagramas. Esta falta de carga semântica faz com que a fase de análise e captura de requisitos deposite demasiada importância em descrições efectuadas em língua natural, ou então, em informação implícita e não comunicada no modelo.

O processo apresentado neste capítulo pretende aumentar o grau de rigor da descrição dos casos de uso e torná-los o elemento central da análise de um sistema. A estratégia traçada tem três vectores que a justificam:

- o reforço da componente de descrição dos casos de uso por inclusão de expressões rigorosas;

- a construção de uma abordagem unificada com recurso a várias vistas do mesmo componente, como mecanismo de consolidação semântica da modelação, e
- uma preocupação em permitir a operacionalização do processo através do recurso a prototipagem.

Este capítulo apresenta um conjunto de práticas que devem ser seguidas na fase de análise de modo a aumentar a qualidade da modelação e, espera-se, igualmente do sistema final.

4.2 Método de modelação

O desenvolvimento de um sistema software não é actualmente uma tarefa simples, na medida em que a dimensão e requisitos dos problemas não permitem que tal se aborde sem rigor. Não basta ter uma ligeira e superficial descrição do sistema, antes deve o engenheiro de software estar preparado para uma escala de complexidade que só pode ser dirimida com o recurso a metodologias e técnicas bem fundadas.

Se a aplicação de um processo e uma metodologia é importante em problemas de pequena ou média dimensão, em sistemas complexos de larga escala ela é ainda mais crítica. Esta criticidade deriva do facto de se ter que lidar com a complexidade de forma controlada e através da decomposição desta, sustentadamente se decompor o problema em vistas diferentes, permitindo conceptualizar o sistema antes de se entrar na fase de desenvolvimento de código. Além das vantagens enunciadas, e que são visíveis na construção e instalação do sistema, a adopção de um processo rigoroso permite também usufruir de algumas melhorias durante o restante tempo de vida do sistema. É reconhecidamente mais simples efectuar alterações e melhorias num sistema bem construído, com uma boa modelação, do que num sistema que não possua estas características.

Se a importância de uma metodologia e respectivo processo parece ser evidente, convém também não confundir com o sistema os modelos que na sua aplicação se produzem. Os modelos são apenas uma forma de conceptualização do sistema final, porventura na maior parte dos casos sem este estar construído. O esforço para a construção de um modelo que permita suportar as fases de análise e concepção não pode desviar a atenção do analista de que um modelo é apenas uma fase intermédia na construção de um sistema. Não se deve também esquecer que o cliente do sistema não valoriza o modelo, pretendendo ter fundamentalmente o sistema final com o comportamento desejado. O modelo deve ser apresentado como uma forma de comunicação com

o cliente e com os utilizadores do sistema, permitindo obter o máximo de informação sobre este. Sempre que possível, o modelo deverá poder ser operacionalizado, permitindo que esta comunicação com o utilizador seja feita através de um protótipo, possibilitando inclusive a recolha de informação sobre requisitos não funcionais.

O levantamento de requisitos e as tarefas de análise têm de ser consideradas actividades fundamentais no processo de construção do sistema. É possível referir que parte substancial dos desvios no planeamento de um projecto de software tem como causa (ou uma das principais causas) um fraco empenho nestas fases do processo [Molkken 03]. À medida que o sistema a construir, ou apenas a manter, apresenta características de complexidade evidentes (quer estas sejam intrínsecas ou decorrentes do modelo de negócio existente), a importância da fase de levantamento dos requisitos é acrescida. É normal associar as metodologias a tipos particulares de sistemas, considerando que para um determinado tipo de sistema existe uma metodologia mais adequada. No entanto, a necessidade de metodologias e processos parece ter menos a ver com o tipo de sistema que se quer construir mas mais com a complexidade do mesmo.

Note-se que à medida que a complexidade do sistema aumenta, quer esta seja função dos requisitos funcionais quer seja derivada do ambiente de instalação e execução, mais importante se torna a existência de metodologias e modelos de processo que privilegiem o reforço das fases anteriores ao desenvolvimento. É, na perspectiva do autor, errada a assumpção, mais ou menos generalizada, de que a complexidade de construção de um sistema radica, na sua maioria, nos aspectos que mais têm a ver com as fases de desenvolvimento e elaboração. À medida que o sistema software é mais complexo, mais importância deve ser dada às fases de levantamento de requisitos e análise, de modo a que o investimento humano e tecnológico a ser realizado na construção do sistema seja devidamente direccionado.

Mais importante que bem construir um sistema software, parece ser o facto de se construir o sistema software desejado pelo cliente. Este facto, facilmente entendível, leva a que seja relevante que as tarefas de levantamento de requisitos e posterior análise devam ser mais valorizadas. A interacção inicial que os clientes têm com a equipa de projecto (ou pelo menos com alguns perfis da equipa de projecto) e que conduz ao levantamento e elaboração dos casos de uso, deve ser reforçada, quer no aspecto quantitativo como no qualitativo, de forma a que se consiga que os modelos que vão ser desenvolvidos apresentem características que permitem afirmar que, de facto, correspondem ao sistema software que o cliente pretende.

O reforço, qualitativo e quantitativo, que se pretende atribuir a esta fase de análise é consequência directa da importância que este trabalho atribui à existência de um

modelo de processo mais detalhado e rigoroso, que implique que a equipa de projecto despenda mais esforço na fase inicial, mas que se acredita que pode melhorar a qualidade final do produto entregue. Esta qualidade acrescida não se pode medir em termos de qualidade intrínseca do desenvolvimento (embora aqui também se esperem melhorias), mas sim em função do aumento de rigor na qualidade da análise e no reforço do processo de construção do sistema, que se traduz na evidência que os requisitos do cliente são bem percebidos e modelados e que existe da parte do cliente a consequente validação da interpretação que a equipa de projecto construiu a partir dos requisitos expressos. Este aumentar de complexidade e rigor no processo de construção dos sistemas software, com o objectivo final de aumento da qualidade, é algo que é comum às empresas sob a forma dos processos de certificação da qualidade, quer estes sejam os ISO [Paulk 94] ou o CMMI [Chrissis 06], que são direccionadas actualmente aos fornecedores de soluções de software.

Este acréscimo de complexidade, quer nos requisitos quer nos métodos de construção da solução, é um aspecto essencial para que as metodologias e processos de desenvolvimento utilizados na construção de um sistema software sejam devidamente valorizadas e consideradas. Note-se que o cliente tem, regra geral, uma relação de menor empatia com os processos, sendo mais atento ao resultado final e à capacidade deste corresponder aqueles que eram os seus requisitos iniciais. Usualmente o que é importante para o utilizador final é o resultado e a forma como este se integra no processo de negócio que está a suportar. Para o engenheiro de software que faz parte da equipa, o processo deve ter uma valoração importante, na medida que é ele que vai, em bom rigor, influenciar o resultado final e a avaliação por parte do cliente, do grau de sucesso do sistema.

Das metodologias referidas no capítulo 2, as metodologias orientadas aos objectos parecem ser aquelas que melhor lidam com a complexidade inerente dos sistemas. Primeiro, porque o modelo dos objectos é aquele que melhor se adequa à complexa interacção das entidades dos sistemas actuais, permitindo de forma natural descrever comportamentalmente as diversas formas de comunicação inter-objectos ou inter-componentes. Segundo, porque permite que a evolução do sistema se possa fazer de forma controlada e localizada (orientada ao objecto ou componente), que se quer modificar, sem comprometer o resto do sistema. Terceiro, porque a forma como o paradigma define os objectos possibilita que cada entidade possa ser desenvolvida de forma independente do contexto, o que é fundamental, pois limita interferência.

A utilização de metodologias de desenvolvimento orientadas aos objectos permite que se possa utilizar como linguagem a UML, tirando partido do facto de esta ser

uma linguagem standard OMG, e de ser transversal a muitas ferramentas, quer de modelação quer de desenvolvimento.

Do ponto de vista dos conceitos e propósitos da Engenharia do Software é importante que a metodologia em que nos baseamos seja rigorosa e permita a criação de modelos sobre os quais seja possível raciocinar formalmente. Surge assim a necessidade de incorporar no processo técnicas rigorosas conjuntamente com outras técnicas, chamemos-lhes informais, permitindo desta forma o colmatar de eventuais falhas no que respeita a rigor e precisão. Neste trabalho, procura-se a integração destas técnicas rigorosas com as técnicas informais existentes e amplamente utilizadas de modo a proporcionar maior rigor e precisão ao processo de análise.

Pretende-se que o modelo de processo utilizado seja evolutivo, na medida em que o rigor utilizado em partes do modelo possibilita a operacionalização dos mesmos e consequente prototipagem. É importante que estas alterações aos modelos de processo utilizados sejam validadas pelos potenciais utilizadores e devem apresentar um conjunto de actividades que façam sentido e que correspondam à criação de elementos do modelo que possam ser discutidos pela equipa de projecto (os *deliverables*). Um modelo de processo, bem como uma metodologia, só tem a aceitação da comunidade científica e dos utilizadores industriais se identificar correctamente as suas etapas e o resultado das mesmas, permitindo também que se possa proceder à interacção entre artefacto das diversas etapas.

Deve ser ainda referido que um novo modelo de processo não é uma solução imediata para todos os males que afectam a construção de sistemas software, que deve ser seguida de forma estrita, e que conduz, sem falhas, ao resultado pretendido. É sabido que a construção de sistemas software complexos, dada as suas características intrínsecas, recorre a um modelo de processo tipicamente iterativo e incremental e que por vezes pode não seguir fielmente as etapas do processo subjacente. A experiência da equipa de projecto e a avaliação que faz em cada uma das diferentes etapas permite que se possa, e deva, aquilatar do andamento do processo de modo a ser possível a intervenção dos engenheiros de software responsáveis (e por vezes do próprio cliente). Uma das vantagens das ferramentas de modelação existentes (Visual Paradigm, Poseidon, Borland Together, etc.) reside no facto de não estabelecerem um modelo de processo rígido, possibilitando a sua adequação a diversas metodologias e modelos de processo.

Em jeito de conclusão, deve ser referido que o modelo de processo que este trabalho apresenta tem sido utilizado numa perspectiva académica e apresentado a alunos finalistas dos cursos de Engenharia de Software. A utilização do modelo de processo é feita

a esse nível no intuito de fornecer uma maior componente de análise, com realce para o papel estruturante dos casos de uso como mecanismo básico de recolha e análise de requisitos. Tendo em conta que os alunos que frequentam o curso possuem valências técnicas a nível do modelo de programação orientado aos objectos e são portadores de conhecimentos que os habilitam a desenvolver tecnicamente sistemas software complexos, a introdução deste modelo de processo permitiu que, previamente às tarefas de concepção e elaboração, adquirissem competências na fase de análise. A utilização dos casos de uso, numa estratégia *Use Case Driven* [Cockburn 01], permite completar um ciclo, uma iteração, de análise e desenvolvimento das suas descrições e das condições que regem o ciclo de vida das entidades determinantes do sistema. Conjuntamente também se utilizou a construção dos diagramas de estado para a descoberta da interacção com o utilizador, sendo feita a modelação do controlo de diálogo logo após a fase de análise, trazendo benefícios para a elaboração e concepção da arquitectura de classes e permitindo ao cliente a validação da interface e dos resultados pretendidos.

A Figura 4.1 permite apresentar de forma visual as áreas fundamentais que sustentam o modelo de processo que é proposto.

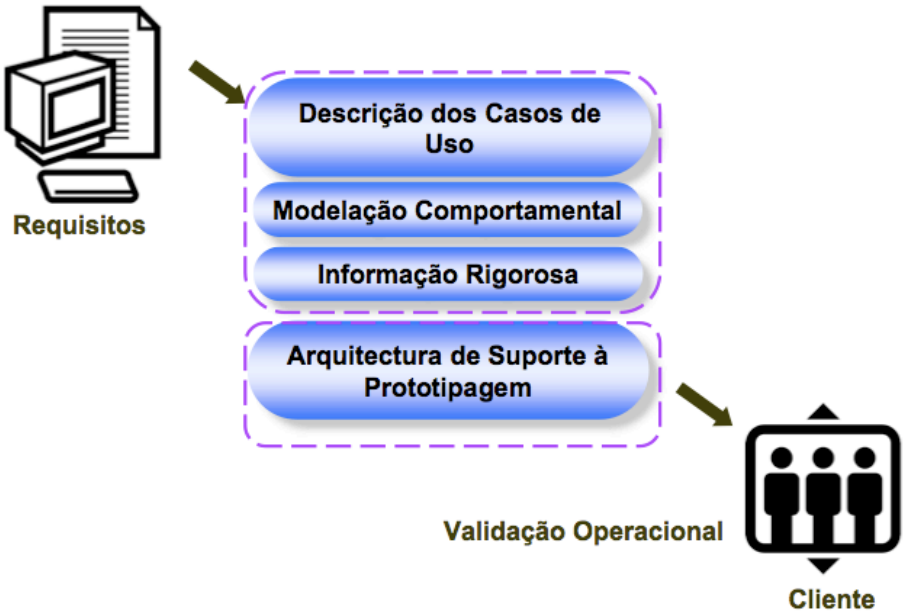


Figura 4.1: As componentes do modelo de processo proposto.

A ideia principal deste modelo de processo passa por garantir que os documentos de projecto que são originados de forma informal pelo cliente, sejam convertidos em

peças de modelação mais completas e rigorosas que possibilitem a correcta sustentação das tarefas de desenvolvimento do sistema. Os requisitos são convertidos em casos de uso e são convenientemente descritos com a ajuda do cliente e seguindo um arquétipo de descrição bastante descritivo.

Para todos os casos de uso do sistema são criadas descrições do seu comportamento recorrendo à utilização de outras peças de modelação. Desta forma obtém-se uma especificação completa dos requisitos e qual é o seu modelo de comportamento.

A última peça de modelação que é proposta, consiste na escrita rigorosa das restrições que existem à execução do caso de uso, tendo em conta a sua descrição e a modelação comportamental que foi feita. A captura desta informação rigorosa é importante para tornar a informação que o cliente possa ter e não a tenha transmitido convenientemente nos requisitos. É também valiosa porque é escrita numa forma que é útil para a equipa de projecto que conduzirá as fases de concepção e desenvolvimento.

Todas estas peças de modelação contribuem para que o cliente possa validar operacionalmente os requisitos que foram capturados pela equipa de analistas. É possível animar as especificações de comportamento possibilitando dessa forma, que o cliente possa pronunciar-se sobre a qualidade de informação que se obteve e prototipar a execução dos requisitos funcionais.

4.2.1 Apresentação do Modelo de Processo

O modelo de processo que rege a modelação de sistemas complexos deve garantir que:

1. exista uma sobre-especificação rigorosa da componente de análise de requisitos;
2. possibilite uma descrição multi-vista coerente do sistema, e
3. permita uma abordagem operacional dos modelos.

A sobre-especificação da componente de análise de requisitos, permite que os modelos possam ser mais ricos e possibilita que o processo de recolha de requisitos seja mais exaustivo. A recolha de requisitos e identificação dos cenários passa a incorporar toda a informação que o cliente possa implicitamente ter sobre a execução de um determinado caso de uso. A identificação do fluxo de controlo num caso de uso é explorada de forma a que seja possível a descrição do comportamento pretendido, em termos de eventos de transição, de actividades executadas e de alterações ao estado do sistema. É comum que os utilizadores expressem as funcionalidades que esperam encontrar no

sistema sob a forma de tarefas, isto é, o conjunto das actividades e acções que levam à obtenção de algo que está estabelecido como um objectivo da utilização do sistema.

Esta sobre-especificação possibilita a captura de informação sobre a coerência de estado dos objectos participantes de um caso de uso, permitindo incorporar no modelo, o mais cedo possível, informação que só era adicionada em estágios posteriores.

A captura de mais informação (e mais diversificada), nesta fase, permite que se construam outras vistas do modelo. Este é um processo que vai sendo refinado até que se obtenha um modelo completo e válido do sistema. A necessidade de organizar a informação que é recolhida na descrição dos cenários dos casos de uso determina que se construam outros modelos comportamentais com essa informação.

A notação para a construção destes modelos deve ser diagramática, possibilitando uma fácil leitura por parte da equipa e do cliente. Tendo em conta que na criação de um modelo de um sistema complexo existe muita informação que é transmitida e deve ser representada e que essa informação é de índole distinta, faz todo o sentido que se privilegie uma notação que seja multi-vista ou multi-diagrama. A utilização de representações gráficas são indicadas para representar os modelos de sistemas interactivos e reactivos, nos quais se tornam especialmente úteis pois permitem identificar de forma inequívoca os eventos desencadeados pelos actores e as respostas (e transições) do sistema. Essa representação gráfica deve permitir a comunicação entre a equipa de projecto e o cliente, e tem como objectivo último a cristalização da informação que é descoberta na fase de análise e que, descrita numa linguagem gráfica, vai ser transmitida à equipa de projecto e vai ser um instrumento facilitador na transição da análise para a concepção e posterior desenvolvimento.

Neste trabalho, utiliza-se a linguagem UML pois esta disponibiliza uma série de vistas (diagramas) que possibilitam a abordagem a um sistema por várias vertentes. A UML é independente do processo utilizado, embora a sua génese radique nas metodologias orientadas aos objectos e esteja especialmente vocacionada para ser utilizada por modelos de processo com cariz incremental e iterativo e com uma abordagem baseada em casos de uso [Jacobson 92, Rosenberg 07].

A possibilidade de operacionalização é também um requisito essencial, na medida em que se pretende ter o mais cedo possível no processo a capacidade de prototipagem e de validação de algumas assumpções. Como foi também visto no capítulo 2 a evolução de um modelo faz-se, por vezes, validando através de um protótipo os requisitos capturados com o cliente. O protótipo pode inclusive servir de mecanismo de descoberta de novos requisitos. A sobre-especificação dos casos de uso permite recolher informação que possibilita a criação de um protótipo, bem como possibilita que essa informação,

por ser rigorosa, possa ser matematicamente trabalhada.

Esta capacidade de operacionalização permite que a equipa de projecto, junto do cliente, possa validar a informação que foi recolhida até ao momento, possibilitando numa fase inicial do projecto a existência de realimentação ao modelo que está a ser construído. O modelo de processo que se adopta advoga a possibilidade de serem cometidos erros, quer de modelação quer de interpretação dos requisitos, nas diversas fases de análise. É uma vantagem de um modelo de processo o facto de não ter uma estrutura rígida e dessa forma permitir que sejam revisitadas fases anteriores com o objectivo de acrescentar informação que entretanto se descobriu, ou então com o propósito de corrigir informação que tenha sido erroneamente introduzida nos modelos. Os modelos de processo muito rígidos são de difícil aplicação tendo em conta a heterogeneidade de tecnologias actualmente existentes e os diversos tipos de sistemas que se podem com elas construir.

É natural que não seja possível ser absolutamente estrito na separação entre as diversas fases (como se advoga no modelo em cascata), mas que em função da complexidade do sistema a construir seja necessário que o processo possua características que facilitem sucessivos momentos de iteração e inclusive o recuo para fases anteriores. Esta iteratividade resulta muitas vezes dos testes conduzidos em fases iniciais do projecto, que possibilitam uma atempada realimentação (e manutenção) dos requisitos. Em modelos de processo mais estritos e rígidos, as tarefas de teste apenas surgem mais tarde e já em fase de construção do sistema, sendo nessa altura mais difícil lidar com eventuais situações que demonstrem uma fraca captura dos requisitos. A utilização de protótipos, com a consequente validação dos modelos desenvolvidos nas fases iniciais do projecto de construção do sistema software, permite identificar e corrigir erros que de outro modo apenas seriam detectados na fase de testes posterior ao desenvolvimento, ou no pior cenário, apenas em tempo de funcionamento.

Os modelos de processo, por mais bem definidos que sejam e por mais que estejam dotados de boas ferramentas para a construção dos modelos, não propiciam de forma autónoma uma garantia de qualidade do produto final. É condição essencial que exista um esforço de adaptação às necessidades do sistema software a construir e às características da própria equipa de projecto. Parte-se do princípio que o cliente é uma parte importante da equipa de projecto e que tem a capacidade para participar na fase de análise, através da identificação dos requisitos funcionais, e não funcionais, do sistema e das restrições que se devem aplicar em tempo de funcionamento. O cliente deve ser capaz de expressar, ainda que não de forma necessariamente formal, as condições em que as funcionalidades do sistema estarão disponíveis, bem assim como deve ser capaz

de transmitir à equipa de projecto as regras que determinam o invariante de estado do sistema.O que se espera é que o cliente seja um especialista no modelo de domínio do sistema que se quer construir (ou efectuar manutenção evolutiva ou correctiva), mas ao qual provavelmente lhe faltam competências em termos de tecnologia e modelos de programação. No entanto, com a ajuda da equipa de projecto deve ser capaz de identificar os requisitos e expressá-los de forma concreta, permitindo aos engenheiros de software a formalização desses mesmos requisitos em modelos UML anotados com expressões rigorosas.

4.2.2 A construção do Modelo multi-vista

A forma como se constrói o modelo do sistema deve ser feita de modo que à medida que se vai tendo mais informação sobre o sistema o modelo possa ser refinado. Não faz sentido que a filosofia de criação de um modelo multi-vista, com a utilização da UML, obrigue a que cada uma das vistas seja completamente terminada antes de se passar para a vista seguinte. O processo deve ser naturalmente iterativo, indutor de refinamento sucessivo e com realimentação, seguindo um padrão semelhante ao modelo de processo em espiral [Boehm 88]. É também útil que esta interactividade se produza na construção do modelo, na medida em que a descoberta numa fase posterior da falta de alguma informação, pode originar que se tenha de recomeçar o processo para determinada(s) entidade(s) ou vista(s). Por vezes, a modelação com o recurso à utilização da UML é vista como sendo aproximada do modelo em cascata, ou seja, os diagramas são produzidos por uma ordem específica e num determinado diagrama tem-se uma vista do problema. Existem diversas abordagens na literatura que sugerem esse *modus operandi*, esquecendo que tal abordagem é contrária à própria filosofia das metodologias por objectos e não se enquadra no que é definido pelo Unified Process [Jacobson 99, Software 98]. Nessas perspectivas mais rígidas é usual apresentar o processo de construção do modelo como sendo sequencial, sendo que uma vista se sucede à outra e à medida que se avança o conhecimento sobre o sistema aumenta e vai ficando mais estável. Embora este acréscimo de conhecimento efectivamente se verifique, não parece ser possível construir o modelo correcto de um sistema sem o recurso a iterações sucessivas com os respectivos passos de refinamento.

É comum encontrar-se a especificação do processo de modelação como sendo a sucessão dos seguintes diagramas:

1. Diagrama de casos de uso - onde se levantam todos os casos de uso e eventualmente se faz a descrição deles de forma informal (ou apenas bem organizada

numa sintaxe pré-estabelecida) em linguagem natural. O diagrama de casos de uso pode ter diversos níveis de abstracção[1] sendo que nele são identificados os casos de uso, os actores e espera-se os diversos cenários de utilização;

2. Diagrama de classes - onde se caracterizam todas as entidades que constituem o domínio da aplicação, isto é, aquelas que fazem parte do domínio do problema e aquelas que são tecnologicamente necessárias para a concretização do sistema. A título de exemplo as primeiras podem mapear conceitos como Encomenda, enquanto que um exemplo das segundas pode ser concretizado em entidades como LinhaEncomenda. Podem existir diversos graus de complexidade no diagrama de classes, desde o diagrama apenas com as entidades principais, até ao diagrama mais próximo da concepção arquitectural e de implementação.

   É desejável que se possa completar o diagrama de classes com a informação sobre os relacionamentos entre as classes existentes e sobre o estado e comportamento de cada uma das classes. No fim desta fase da modelação existe uma definição de qual é a arquitectura do sistema software e de quais os padrões de software que faz sentido aplicar;

3. Diagrama de estado - onde para cada uma das entidades relevantes[2] se efectua uma análise sobre as transições que sofre e pelos estados porque passa. No diagrama de estados utilizam-se as operações especificadas na parte comportamental da entidade, conforme o diagrama de classes;

4. Diagrama de sequência - onde se detalha para cada uma das acções do sistema quais os objectos envolvidos e qual a troca de mensagens que existe para a pretendida prossecução da acção. Nesta fase começa a ser mais difícil ter a percepção do que se modela, isto é, quem alimenta este processo, o diagrama de classes onde existe detalhada a colecção de operações das entidades, ou a informação dos casos de uso relativa aos fluxo de controlo do sistema;

5. Diagrama de actividades - onde se especifica o controlo de fluxo possível de inferir a partir dos casos de uso e dos requisitos do cliente. Este diagrama é dirigido à modelação de fluxo que descreve a parte comportamental a um nível mais abstracto, e

[1] Favorecidos inclusive pelas ferramentas software de suporte à construção do modelo.

[2] Esta atribuição de relevância a uma entidade corresponde na maior parte das situações ao facto de ser uma entidade do domínio do problema que também faz parte do modelo arquitectural.

6. Outros diagramas, como os diagramas temporais, consoante a dimensão do problema. Existem vistas importantes no que respeita ao nível de instalação e operacionalização do sistema software, bem assim como à arrumação lógica dos diversos componentes.

Esta descrição do processo de modelação, apesar de organizada e de permitir definir os diagramas mais importantes, apresenta algumas lacunas que não passam despercebidas a quem tem experiência na modelação de um sistema software complexo e a quem privilegia uma abordagem mais sustentada e rigorosa do processo de construção de sistemas.

Entre essas dificuldades podem referir-se como mais significativas, as seguintes:

1. não é claro que seja possível a um elemento da equipa de projecto inferir o diagrama de classes a partir do diagrama de casos de uso, pelo menos de forma imediata e não evolutiva. O diagrama de casos de uso identifica as funcionalidades existentes, mas não fornece *per si* informação suficiente (na sua forma original) para que o domínio da aplicação seja conhecido em detalhe e se possa construir o diagrama de classes com todos os relacionamentos necessários. Apesar de ser possível, com mais ou menos dificuldades consoante o sistema, apreender as entidades principais da aplicação, não é fácil a construção do restante diagrama, na medida em que falta informação que não está explícita no diagrama de casos de uso e na descrição destes.

   Não é concebível que a transição entre estes dois diagramas seja tão pronunciada, sendo que faz sentido que o processo de construção do diagrama de classes se faça gradualmente. Procede-se normalmente à divisão dos casos de uso por famílias de funcionalidade, o que permite que se construa uma parte do diagrama de classes, se volte ao diagrama de casos de uso para refinar o modelo, e assim sucessivamente.

   Uma alternativa possível seria a passagem dos casos de uso, para diagramas de objectos, onde se especificaria cada uma das funcionalidades com base nos objectos que nela intervêm. Esses diagramas de objectos dariam depois origem ao diagrama de classes. Desta forma conseguir-se-ia perceber o que aconteceria ao sistema em tempo de execução e identificar-se-iam os objectos e a sua importância para um determinado cenário de utilização.

2. não é claro o papel dos diagramas de sequência após os diagramas de classe. Sendo esta transição bem vincada, então os diagramas de sequência teriam de incidir

necessariamente sobre as operações dos objectos descritas no diagrama de classes. No entanto, faz sentido que os diagramas de sequência sejam utilizados para descrever as interacções existentes nos cenários dos casos de uso. A necessidade de prever uma operação não contemplada na classe do objecto receptor faz com que o diagrama de classes seja alterado e refinado, existindo desta forma uma comunicação e realimentação de informação entre os diagramas de classe, de sequência e mesmo de casos de uso.

3. a modelação da interface com o utilizador segue os mesmos passos de modelação das restantes componentes do sistema, mas apresenta necessidades próprias e constitui-se como um modelo complexo, dentro de um modelo mais lato que representa a totalidade do sistema a especificar. A especificação da componente do controlo do diálogo pode ser feita utilizando os mesmos diagramas, como forma de descoberta dos eventos da camada interactiva. A modelação desta camada de interface com o utilizador assenta principalmente nos diagramas comportamentais, como sejam os de actividade e estado, embora devido à natureza do que se pretende modelar origine diagramas complexos e eventualmente com pouca ligação à modelação da camada computacional.

Pretende-se pois, adoptar um modelo de processo de modelação que possibilite uma evolução do modelo de forma iterativa, começando nos diagramas de caso de uso, e, para cada um deles, ir construindo os diagramas subsequentes. A forma final dos diagramas será o *somatório* dos resultados dados por todas as linhas de contribuição que se formam a partir do diagrama de casos de uso. Isto é, enquanto não se percebe se já se chegou a uma situação em que se pode considerar o modelo como estável e completo, o analista procede a sucessivos passos de refinamento e avaliação do que já obteve.

A Figura 4.2 apresenta de forma visual as diversas linhas de modelação que são criadas a partir de um caso de uso e que conduzem à criação de excertos dos outros diagramas. Como é evidente na imagem, existem diversos níveis de refinamento da informação e cada caso de uso origina um conjunto de documentos (principalmente diagramas) que o vão sucessivamente descrevendo. O processo converge quando de um determinado passo da espiral para o passo seguinte a contribuição começa a ser marginal.

A necessidade de existência de processos de modelação iterativos e flexíveis é uma consequência da experiência prática, porque por vezes a inflexibilidade de um processo pode levar a que ele não seja seguido, resultando, por norma, em modelos incompletos

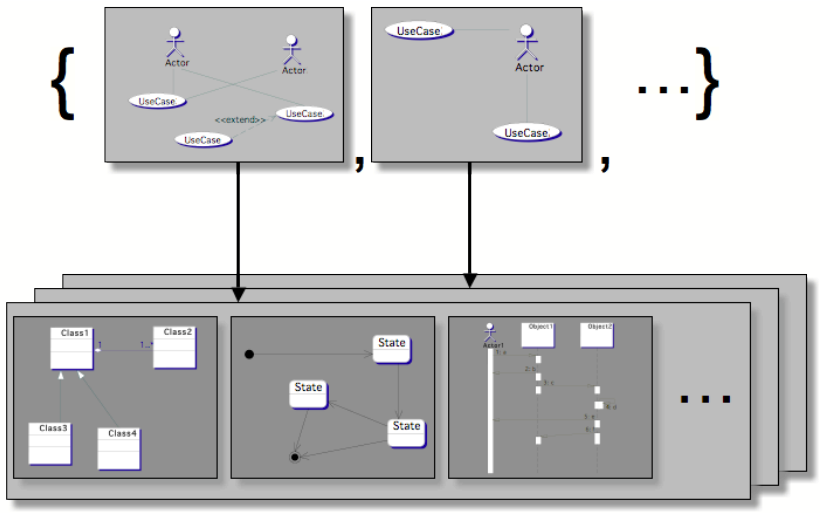


Figura 4.2: O processo evolutivo de criação de modelos.

ou inconsistentes. A incompletude e inconsistência de um modelo é, como se sabe, factor decisivo na qualidade final do produto.

Contudo, é importante referir que uma metodologia de modelação e um modelo de processo não produzem bons modelos de forma autónoma. São apenas um instrumento, tal como os modelos criados, de forma a melhorar, ou aumentar, a qualidade do sistema a ser desenvolvido. A filosofia subjacente a este processo induz também a colaboração do cliente na fase em que essa colaboração assume maior relevância. Com esse propósito, o modelo de processo assume que a intervenção do cliente é mais profícua se permitir que o elemento de granularidade seja um caso de uso e que o detalhar desse caso de uso permita ir anotando o diagrama de estados, sequência e classe.

A Figura 4.3 apresenta a proposta de processo de modelação e consubstancia o processo de realimentação contínuo que se defende. Merece especial relevo a colocação de actividade de teste em todas as fases, como indicador que os elementos resultado de cada uma das fases devem ser validados e caso não sejam considerados merecedores de serem aceites devem voltar a ser trabalhados. Esta assumpção é válida tanto para os documentos de análise como para os modelos arquitecturais ou o código fonte. É um processo assumidamente iterativo em todas as suas fases, e com realimentação para as fases anteriores. Tal como expresso na Figura 4.2, em cada fase a construção dos diagramas é ela também evolutiva.

Note-se que a imagem apresentada na Figura 4.3 apresenta um modelo de processo para a construção integral do sistema. Neste trabalho, as nossas preocupações centram-

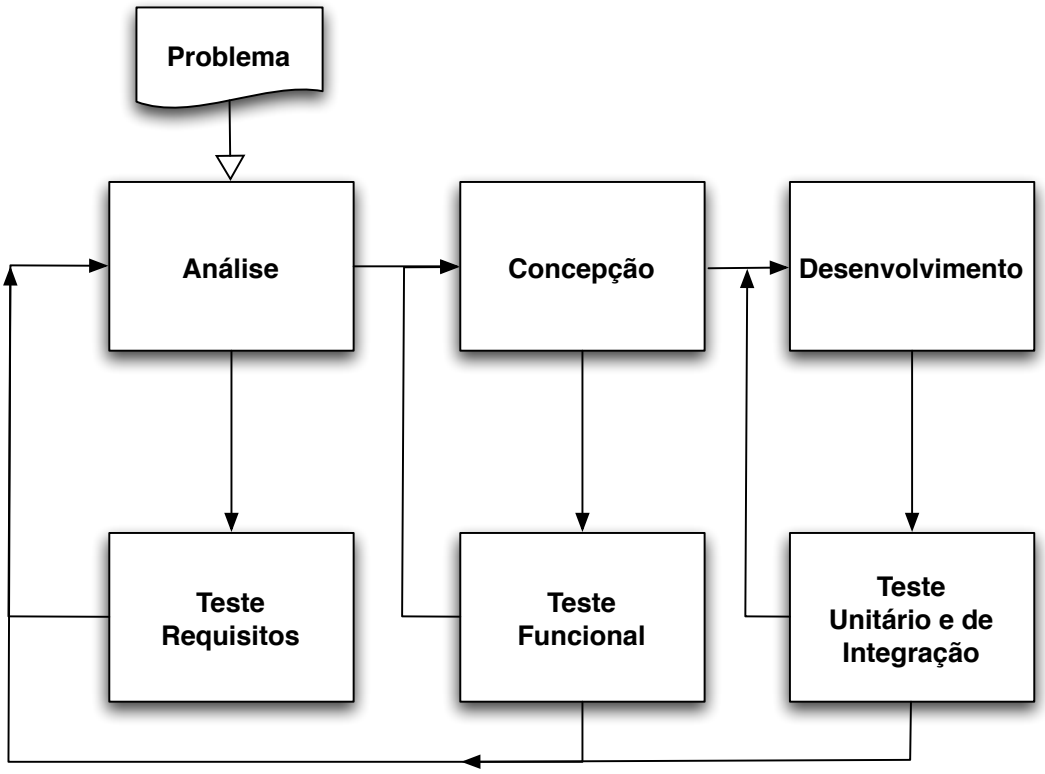


Figura 4.3: Modelo de processo utilizado na construção de sistemas software.

se exclusivamente na fase de análise, sendo que assumimos que as fases seguintes de concepção e desenvolvimento já possuem métodos e processos que estão estabilizados e que dependem em grande medida do modelo de programação utilizado. Desta forma a imagem representa desta forma dois tipos de processos. Um processo mais lato que individualiza cada uma das grandes fases externamente visíveis (análise, concepção e desenvolvimento) e um processo interno a cada uma dessas fases com sucessivos passos de iteração. Este modelo de processo está alinhado com a definição do Unified Process, acrescentando-lhe porém a capacidade de teste dos requisitos, como forma de validação com o cliente da especificação fornecida.

A importância dos cenários

Nos casos de uso, cenários correspondem a sequências de passos que descrevem interacções entre um utilizador (um actor) e o sistema. Um caso de uso é um conjunto de cenários que estão relacionados por um objectivo comum [Fowler 04]. A elicitação dos cenários é uma técnica que está presente na maior parte dos métodos de processo

orientados aos objectos, embora em muitos desses métodos a definição dos cenários seja feita de forma bastante informal. A dependência entre cenários não é modelada em grande parte dos métodos, e mesmo nos diagramas de caso de uso, tal como definidos por [Jacobson 92], essa descrição não existe, apesar de estarem contempladas as dependências etiquetadas como << *extend* >> e << *include* >>. A lógica que sustentou o facto de não se modelarem outro tipo de dependências - como por exemplo a concorrência, restrições temporais, estruturais, entre outras - tem a ver com a necessidade de manter o modelo de casos de uso o mais simples possível. Note-se porém que a descrição dos cenários quando recorre ao uso de linguagem natural herda os problemas típicos desta abordagem, tendo a simplicidade um custo elevado.

A utilização de cenários apenas permite obter descrições parciais do comportamento do sistema, se não for acrescentada informação que possibilite a descrição das dependências entre eles. Dessa forma, recolhem-se detalhes importantes acerca do funcionamento do sistema que devem ser registados e modelados, nomeadamente se esta informação for utilizada para a geração de casos de teste.

A validação e a verificação dos cenários é suportada de duas formas: primeiro, através da sua formalização com utilização de outros diagramas UML, nomeadamente diagramas de estado e de sequência (e actividade) e da definição de restrições ao modelo; segundo, através da utilização de uma plataforma de prototipagem que permite a operacionalização das suas descrições.

A validação pode apenas ser feita pelos utilizadores através de inspecção, o que constitui uma forma eficaz de confirmar que a descrição do cenário é a pretendida. A verificação é conseguida através de formalização, na medida em que os cenários descritos sob a forma de linguagem natural são convertidos para diagramas de estado e refinados a esse nível para encontrar inconsistências, omissões e ambiguidades.

As dependências entre cenários, quando consideradas nas tarefas de análise, são uma mais-valia para o processo de modelação na medida em que acrescentam informação às técnicas usuais de captura de requisitos. Estas dependências, bem como a própria informação dos cenários, pode ser utilizada para as actividades de validação e verificação dos requisitos.

Dependências entre os cenários

Os cenários retirados da especificação dos requisitos e da identificação dos casos de uso podem ter relações de dependência entre eles. Essas dependências podem ser expressas de várias formas:

- um cenário pode ser precedido por um outro cenário;

- um cenário pode correr em paralelo com outro cenário;
- informação, ou dados, de um cenário são necessários para outro cenário;
- um cenário na sua execução invoca um outro cenário, podendo ficar ou não à espera do fim da execução deste;

As dependências entre os cenários podem ainda ser categorizadas como sendo de abstracção, temporais ou causais. As dependências de abstracção correspondem a situações de decomposição hierárquica dos elementos do modelo, sejam estas de generalização ou de composição. As dependências temporais correspondem a relações de sequenciação entre cenários. Estas relações podem ser de precedência estrita entre cenários ou então serem relações de tempo-real. As dependências causais reflectem as situações em que a execução de um cenário depende da execução de um outro, podendo essas dependências serem baseadas nos dados que são trocados ou então na existência de um recurso disponível.

4.2.3 A Fase de Análise

Como apresentado e justificado no Capítulo 2 a fase de análise é de crucial importância no desenvolvimento de um sistema software complexo. É nessa fase que o problema se transforma em requisitos que podem ser modelados, trabalhados abstractamente e onde se produzem as especificações que dão origem aos trabalhos de concepção.

A fase de análise é o primeiro momento em que a equipa de projecto toma conhecimento do problema, e é nesta fase que se começam a recolher os primeiros requisitos e a avaliar da viabilidade de resolução. É também nesta fase que se determinam os fluxos de controlo e os cenários possíveis de traçar a partir de um caso de uso. Na análise são produzidos como peças de modelação, o levantamento dos casos de uso existentes, a sua descrição, a identificação dos cenários, a descrição dos eventos que afectam as entidades (nesta fase de negócio), entre outra informação.

O processo de modelação proposto para a fase de análise consiste nos seguintes passos principais:

1. Análise de requisitos - identificam-se os casos de uso, os actores, obtém-se a sua descrição textual e desenha-se o diagrama de casos de uso;
2. Descrição dos cenários e das tarefas - descrevem-se os cenários associados aos casos de uso. É fornecida uma descrição com base no estado dos objectos que suportam

o caso de uso e são identificadas as restrições à execução desse cenário: condições à entrada, alterações à saída do cenário, consistência interna das entidades representadas;

3. Descrição da actividade/comportamento - onde se cria o modelo de máquina de estados que descreve o fluxo de controlo na execução do cenário do caso de uso. Estes modelos são obtidos por transformação dos casos de uso em diagramas de estado;

4. Acréscimo de Rigor - especifica-se em linguagem formal, OCL, as pré-condições, pós-condições e alteração às variáveis internas dos objectos participantes e ao fluxo de execução dos casos de uso, e

5. Descrição da interacção - onde se cria o diagrama de sequência para cada interacção resultante da execução de um caso de uso, permitindo identificar os objectos envolvidos.

Nos diagramas de estado, ao descreverem-se as transições é possível decorá-las com os métodos que a camada de negócio deverá disponibilizar para suportar o comportamento descrito. Este passo metodológico permite que se vá refinando o modelo e que sejam capturadas as operações que implementarão as acções associadas. Sendo o processo iterativo, através da descrição dos diagramas de transição de estado vai-se completando a informação sobre o sistema e, além de se especificar a componente relativa ao fluxo de controlo, também se vai adquirindo informação sobre a arquitectura ao nível do diagrama de classes e sobre as restrições existentes.

A derivação da lógica de negócio, as operações das classes do diagrama de classes, é obtida através da completude do processo de construção dos diagramas de estado e sequência. A natureza iterativa do processo torna por vezes difícil determinar quando se chega a uma situação estável, sendo que até que esta seja atingida o processo produz uma substancial parte do comportamento que suporta a lógica de negócio. Este método parece ser também a forma mais natural de determinar o comportamento da classe, uma vez que as funcionalidades obtidas por este processo derivam totalmente da análise das tarefas que foram identificadas. Sendo assim, o comportamento obtido corresponde à totalidade (ou quase) do que é requerido ao sistema exibir. Existirão sempre métodos utilitários, ou resultantes de algum processo de *refactoring* ou reordenação do código, mas a componente principal deverá ser encontrada através do modelo de processo apresentado.

As diversas actividades atrás apresentadas, utilizam a UML como linguagem de

modelação e tiram partido do facto de esta ser neutra em termos metodológicos e de processo, podendo ser adaptada da forma que pareça mais conveniente à equipa de projecto. A proposta que neste trabalho se apresenta é semelhante às metodologias orientadas aos objectos mais populares, sendo que se baseia no modelo de processo do RUP [Jacobson 99] e na filosofia mais abrangente da OOSE [Jacobson 92], principalmente no que concerne à importância dada aos casos de uso, tendo em linha de conta os trabalhos de Cockburn [Cockburn 01].

A Figura 4.4 apresenta as diversas fases e entidades que devem ser consideradas neste processo e explicita também os principais documentos (peças de modelação) que são produzidos. Como se constata pela imagem, os conceitos base são os apresentados pelo Unified Process e o processo de modelação que é proposto torna evidente um maior esforço na fase de construção das peças de modelação relativas aos requisitos. Os casos de uso são a peça central da análise de requisitos e da própria fase de análise, e esse esforço não se resume à construção do diagrama, sendo particularmente exigente na vertente de descrição textual, na especificação do comportamento com recurso a outras peças de modelação, e na construção das expressões rigorosas que formalmente documentam as restrições associadas aos mesmos.

Na Figura 4.4 estão evidentes todos os passos do processo que se pretende utilizar. A modelação do negócio explicita quais são as entidades do mundo do negócio e os relacionamentos entre elas dão indicações sobre regras de negócio existentes. Outras regras de negócio são transmitidas à equipa de projecto pelo cliente quando detalha os requisitos. Os requisitos são tratados ao nível dos casos de uso e para cada caso de uso procede-se à criação de uma espiral de refinamento até que se considere que os elementos de descrição do mesmo são suficientemente detalhados. Na imagem o caso de uso "Vender"é identificado e é descrito textualmente de modo a que se apreenda qual é o seu fio de execução. Posteriormente o comportamento do caso de uso é detalhado através de diagramas comportamentais, neste caso através dos diagramas de sequência e de estado. A informação que estes diagramas trabalham é baseada na especificação textual do caso de uso e no conhecimento das entidades de negócio. Como mecanismo de introdução de rigor na captura de requisitos são identificadas as restrições à execução do caso de uso e formalmente descritas. Estes elementos de modelação são relativos à captura de requisitos e constituem o resultado da fase de análise. A fase de concepção baseia-se nesta informação, mas procede às transformações que permitem a passagem para entidades da aplicação. Como é apresentado na imagem, a interacção que é descrita na fase de concepção é mais elaborada do que a feita na fase de análise, acrescentando novas entidades que surgem devido a factores que derivam da construção da melhor solução para o sistema.

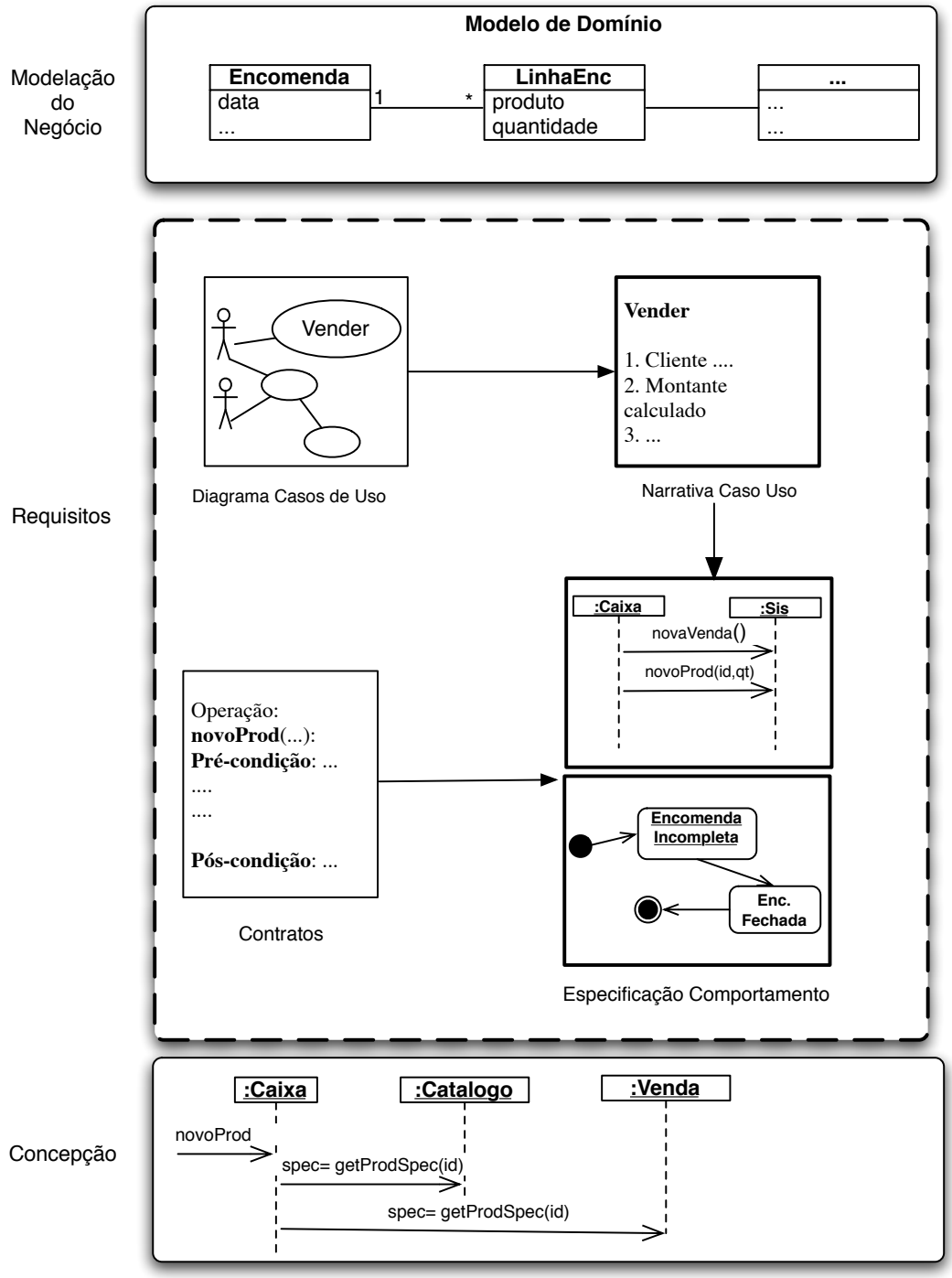


Figura 4.4: A concretização da abordagem orientada ao Unified Process.

A comunicação efectiva entre as três camadas que são evidentes na imagem é muito importante. A informação recebida do modelo de domínio estabelece os parâmetros essenciais da modelação de negócio e normaliza o vocabulário e determina os relacionamentos existentes. O modelo de domínio é um elemento fulcral na definição dos objectos, os seus atributos e relações entre eles. Essa informação é passada à fase de análise de requisitos de modo a ser incorporada nas descrições dos casos de uso e alimenta também as especificações comportamentais que se elaboram, normalizando o vocabulário e unificando conceitos.

A fase de concepção retira também inegáveis vantagens da promoção de uma sobre-especificação da fase de análise de requisitos, visto que ao se estabelecerem os modelos comportamentais e ao se definirem as restrições que os casos de uso apresentam, a quantidade de informação que é transportada para a concepção é consideravelmente superior. Como se demonstrou na Figura 4.4, as definições comportamentais, quer sejam máquinas de estado (diagramas de estado) ou interacções (diagramas de sequência), são documentos que podem ser posteriormente refinados (e optimizados) na fase de concepção, enquanto que as restrições introduzem na equipa de projecto condicionantes à construção do modelo arquitectural e à forma como as interacções são transpostas para o modelo de programação utilizado.

4.2.4 Concepção e Desenvolvimento

Os modelos de processo que se podem aplicar às fases de concepção e desenvolvimento e consequentes melhorias à qualidade do produto daqui resultantes, não são o objectivo primário deste trabalho. As necessidades que uma equipa de desenvolvimento tem durante a prossecução destas fases são de natureza diferente das encontradas durante a fase de análise. Durante a concepção e desenvolvimento a equipa lida com artefactos e problemas substancialmente distintos dos existentes durante a análise, sendo que as soluções que se procuram são normalmente encontradas de um ponto de vista tecnológico ou arquitectural.

Existe no entanto informação que é descoberta durante a fase de análise que é determinante para as fases que se seguem. A identificação de pré e pós-condições e invariantes de estado do sistema, constitui uma peça de informação fundamental para a correcta construção do diagrama de classes e para a passagem a código dos componentes que constituem o sistema. Torna-se assim necessário, associar as restrições identificadas em cada caso de uso aos objectos que o processo de análise identifica como sendo necessários para a execução do caso de uso. A identificação e especificação dos casos

de uso propicia que, numa perspectiva de análise neles centrada, se descubram quais as entidades necessárias para que a interacção que o caso de uso abstrai possa ser descrita. Essas entidades serão potencialmente transformadas em classes do domínio da aplicação, sendo que a informação relativa à formalização das restrições ao seu comportamento devem ser integradas na sua descrição.

Neste trabalho, a operacionalização da validação de requisitos é feita recorrendo a uma arquitectura de entidades descrita em Java, pelo que desta forma pode ser reduzido o salto semântico entre a fase de análise, com a consequente validação do modelo, e a fase de concepção.

4.3 A aplicação do processo à análise

Foram anteriormente referidas quais as diversas fases que neste trabalho se consideram fazer parte das tarefas de análise de um sistema software. Nesta secção, pretende-se, explicitar melhor qual o contributo de cada um dos mecanismos utilizados para a construção do modelo de análise. A justificação desses mecanismos é feita de acordo com os diagramas UML que são originados como documentos de projecto.

A Figura 4.5 apresenta de forma gráfica os resultados que a aplicação do modelo de processo permite obter e que são detalhados nas próximas secções.

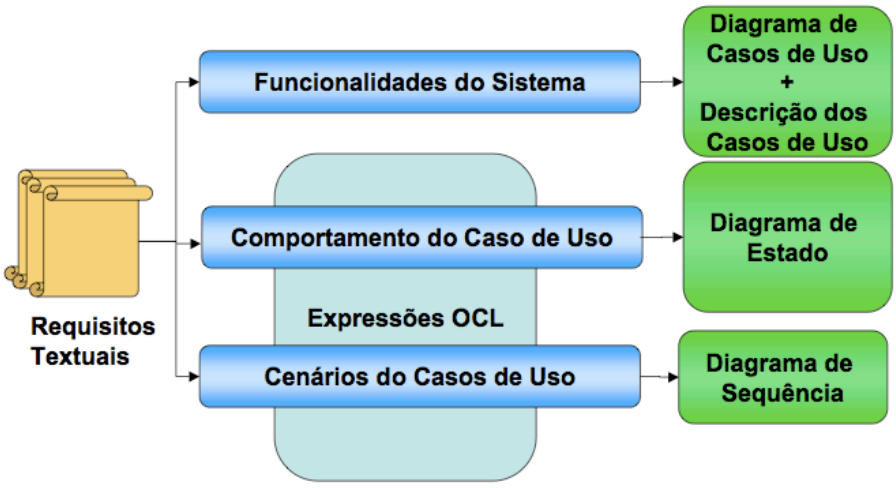


Figura 4.5: Resultados obtidos pela aplicação do processo de modelação.

Os requisitos são o início de todo o processo e são descritos de forma textual com

poucas, ou nenhumas, garantias de rigor. O modelo de processo começa por identificar as principais funcionalidades do sistema e constrói o diagrama de casos de uso e a correspondente narrativa que especifica a alto nível o seu funcionamento.

Uma vez descritos os casos de uso, cada um deles vê o seu comportamento especificado por um diagrama de estados que determina qual é o controlo de fluxo que a sua invocação desencadeia no sistema. Na construção do diagrama de estados são identificadas as operações que suscitam as mudanças de estado e são expressas as condições em que essas transições de estado são possíveis.

O diagrama de estados prevê todas as possibilidades de execução do caso de uso, para o cenário positivo e para os cenários de excepção. Se o diagrama de sequência considerar todos os cenários, torna-se de uma complexidade de representação muito grande, pelo que se aconselha que se desenvolvam diagramas de sequência para cada cenário.

4.3.1 Casos de uso e diagramas de caso de uso

Os casos de uso são uma técnica utilizada para descobrir e descrever o comportamento do sistema da forma como este é visível do exterior. Além de identificarem as funcionalidades que o sistema disponibiliza, permitem também a elicitação dos diferentes tipos de actores envolvidos, e a descrição dos casos de uso a que cada actor tem acesso.

O processo de descoberta dos casos de uso é um processo marcadamente iterativo, sujeito a diversas etapas de refinamento, e que é conduzido pela equipa de projecto em conjunto com o cliente. Pode afirmar-se que o comportamento do sistema é representado pelo conjunto de todos os casos de uso e pela descrição da sua associação aos diferentes actores do sistema. Numa perspectiva influenciada pelas metodologias orientadas aos objectos, é usual que primeiro se identifiquem os diversos actores e depois para cada um deles se determine quais são os casos de uso a que cada um deles tem acesso. Implica esta abordagem que numa primeira fase se pense o sistema a muito alto nível, com um elevado grau de abstracção, procurando identificar os actores e os grandes grupos de funcionalidades.

É usual que no processo iterativo de especificação dos casos de uso o primeiro diagrama que se constrói, por ser bastante abstracto, se designe por *diagrama de contexto*.

Do ponto de vista da representação visual o diagrama de contexto é um diagrama de casos de uso, muito abstracto e simplificado, representando o sistema como uma entidade única com uma fronteira bem definida e rodeado pelos actores que interagem

com ele. Todos os actores são entidades externas ao sistema o que implica que não são responsabilidade da equipa de projecto, quer por serem actores humanos ou por serem representativos de outros sistema já existentes.

Identificar os actores não é uma tarefa fácil e exige iteracção e refinamento, pois a presença de um novo actor pode levar à existência de novas funcionalidades[3]. A descrição do diagrama de contexto, como primeiro passo de construção dos diagramas de caso de uso, é o momento em que se coloca a questão sobre quais os actores existentes e se estes são actores humanos, ou então, outros sistemas que interagem com o que se está a modelar. Em relação a esta questão, o analista tem de tomar a decisão sobre o nível de detalhe que considera quando os actores são sistemas externos.

Nem sempre o diagrama de contexto torna evidente se a comunicação entre o sistema em análise e actores, ou sistemas externos, deve ser representada. Como possível solução para esta questão quatro abordagens podem ser identificadas:

- mostrar todas as associações;
- mostrar apenas as associações relativas a interacções iniciadas por sistemas externos;
- mostrar apenas as associações relativas a interacções em que é o sistema externo o interessado no caso de uso, e
- não mostrar associações com sistemas externos.

Se a solução passar por mostrar todas as associações, todos os sistemas externos que interagem com o sistema em análise são apresentados como actores e todas as interacções são representadas nos diagramas. Esta opção é no entanto, demasiado abrangente, porque em muitos casos existem interacções com outros sistemas apenas por razões de implementação e não por se tratarem de requisitos do sistema, expressos pelo cliente.

No caso em que apenas se identificam as associações relativas à interacção iniciada por sistemas externos, só são representados como actores os sistemas externos que iniciem diálogo com o sistema em análise, o que ainda se revela como muito abrangente, visto que podem existir interacções que não façam sentido para a vista do sistema que a equipa de projecto pretende ter.

[3]A eliminação de actores, por seu lado, leva a que as funcionalidades apenas associadas a estes também deixem de fazer parte do sistema.

A solução mais equilibrada parece acontecer quando são representadas apenas as associações em que é o sistema externo o interessado. Neste caso só são apresentados como actores os sistemas externos que necessitam de funcionalidade fornecida pelo sistema em análise. Do ponto de vista das tarefas de análise não existe perda de informação e a equipa de projecto consegue tipificar (através de *user profiling*) todos os interessados na utilização do sistema, quer estes sejam humanos ou outros sistemas.

Não mostrar as associações com sistemas externos, o que significa que apenas os utilizadores são actores, leva a que a modelação possa ficar incompleta. Nesses casos, e com esta solução, quando existem sistemas externos apresentam-se os seus actores em diálogo directo com o sistema a ser modelado. De uma outra forma, esta solução também é demasiado abrangente e pode levar a confusão sobre quem está realmente a utilizar o sistema.

Depois de descrito o diagrama de contexto, o processo de construção dos casos de uso, na sua vertente diagramática e respectiva especificação textual, segue um processo típico de refinamento. Começa-se por identificar os casos de uso mais genéricos e com menos detalhes e depois progressivamente, à medida que se vai tendo conhecimento mais aprofundado do sistema, constroem-se os casos de uso mais específicos. Quando o diagrama de casos de uso começa a tomar proporções de alguma complexidade, com muitos casos de uso representados e com vários actores envolvidos, como forma de contornar essa mesma complexidade, o analista deve:

- utilizar um mecanismo sintáctico de agrupamento de vários casos de uso dentro de um outro mais genérico, ou então recorrer à utilização de pacotes (*packages*). Esta solução implica que cada um desses agrupamentos será detalhado num outro diagrama, sendo que é usual que os ambientes de modelação forneçam estes mecanismos, e

- efectuar a construção dos diagramas em função dos actores e criar um diagrama de casos de uso por actor. Consegue-se desta forma tornar ainda mais evidente quais são as associações existentes entre determinado actor e as funcionalidades do sistema.

No exemplo da Figura 4.6 o diagrama de casos de uso apresentado é um diagrama de elevado grau de abstracção, no qual está presente o sistema com a sua fronteira evidenciada. Tudo o que está fora da fronteira não faz parte do sistema e não é da responsabilidade da equipa a sua especificação ou construção.

Os casos de uso apresentados são bastante genéricos o que indicia que o diagrama pode apresentar um maior grau de detalhe. Informação dos casos de uso incluídos não está presente, bem assim como não estão identificadas outras relações entre os casos de uso. De forma a não tornar o diagrama de casos de uso uma peça de documentação bastante complexa, é possível dividi-lo e criar tantos diagramas de casos de uso quantos os actores existentes. Quando a ferramenta de modelação suporta o aninhamento e navegação entre diagramas de casos de uso, o analista pode escolher qual é o nível de detalhe a que pretende trabalhar. No caso da imagem, dará origem a um diagrama para o actor Cliente, outro para o Fornecedor e um outro para o Banco. O diagrama apresentado, existindo essa decomposição dos diagramas, é uma aproximação ao diagrama de contexto, pelo nível de abstracção a que se posiciona.

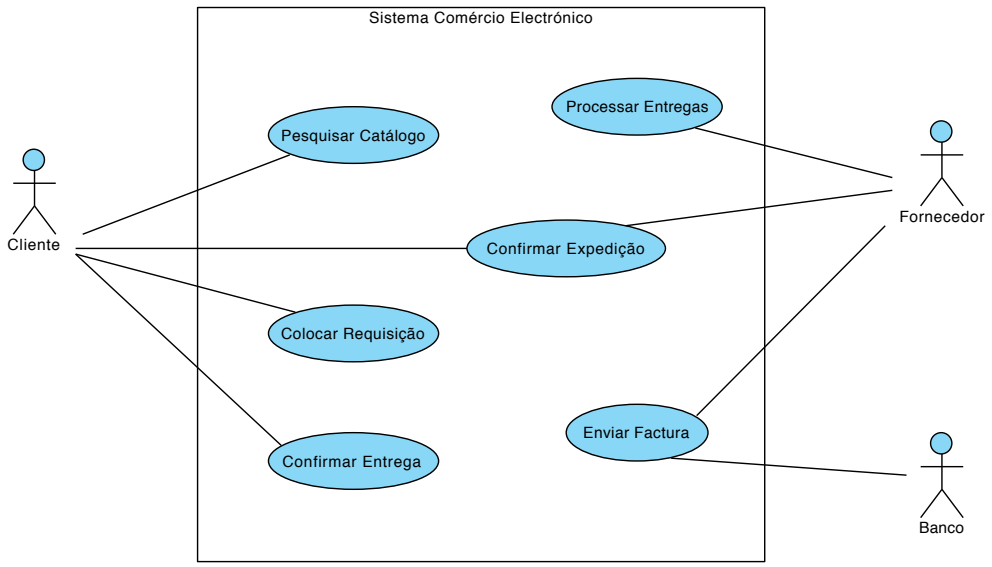


Figura 4.6: Exemplo de um diagrama de casos de uso.

Neste processo de refinamento dos casos de uso o analista deve detalhar até que chegue a uma situação em que o caso de uso em questão já represente uma funcionalidade disponível no sistema. Idealmente, este ciclo de iterações pára quando se chega a um caso de uso que é invocado, no sistema software, pelos actores que a ele têm acesso.

Quanto à natureza dos casos de uso e à sua granularidade, Cockburn [Cockburn 01] identificou três tipos distintos de níveis de abstracção: *summary level*, que corresponde à elicitação de casos de uso com uma grande abstracção numa perspectiva muito geral e difusa, os *user-goal level*, que são aqueles que têm correspondência na visão que o utili-

zador tem das funcionalidades do sistema e, por fim, os *sub-function*, que correspondem quase sempre a operações, e que são os passos algorítmicos em que se dividem os casos de uso do tipo *user-goal level*. Uma regra, derivada do bom senso e da aplicação das boas práticas, diz que não se deve lidar com os casos de uso do último tipo, por serem demasiado específicos e poderem não corresponder aos requisitos do utilizador (mas a uma decomposição destes) nem serem visíveis na camada interactiva do sistema. É preferível a utilização e consequente especificação dos casos de uso que têm correspondência com as acções dos utilizadores, na medida em que assim é mais provável que possa existir diálogo com o cliente na concretização dos modelos que detalham o seu comportamento.

Os diferentes cenários correspondem assim a distintas interacções que um utilizador pode ter com o sistema, podendo o analista registá-los recorrendo a diagramas que capturem interacções. Em UML os diagramas que permitem descrever as interacções são os diagramas de sequência, ou os de comunicação, anteriormente chamados de colaboração. Estes dois tipos de diagramas representam a mesma informação, de forma visual distinta, sendo que as ferramentas de modelação possibilitam a transição automática de uns para outros. Neste trabalho, a descrição dos cenários será feita recorrendo aos diagramas de sequência, visto apresentarem em relação aos diagramas de colaboração vantagens evidentes no que respeita à sua representação gráfica e à capacidade de expressividade que actualmente suportam.

Os diagramas de sequência apresentam como principal vantagem a disposição visual, permitindo claramente identificar a ordem do envio de mensagens entre os diversos intervenientes na interacção, enquanto que nos diagramas de colaboração tal só é possível através da leitura da numeração associada às mensagens. Num diagrama de sequência a ordem das mensagens é dada pela sua posição relativa em relação às outras mensagens, logo não está sujeita a uma numeração como acontece nos diagramas de colaboração. Por último, com a introdução do UML 2.0 não ficou claro como é que nos diagramas de colaboração se representam as *frames* de interacção que existem nos diagramas de sequência, pelo que, aparentemente, estes oferecem menos capacidade de expressão.

Categorias de casos de uso

Depois de identificados os casos de uso é importante reflectir sobre o tipo de funcionalidade que os mesmos encerram. Nem todos os casos de uso correspondem a funcionalidade que altera o estado do sistema, sendo que alguns dos casos de uso correspondem

a operações que interrogam o sistema e devolvem resultados.

As funcionalidades que o sistema exibe podem ser categorizadas em casos de uso que (i) constituem alterações ao estado do sistema e (ii) casos de uso que apenas interrogam o sistema. Exemplos de casos de uso do primeiro tipo são, por exemplo, *"Requisitar Livro"* ou *"Devolver Livro"* num sistema software de gestão de reservas de livros de uma biblioteca, enquanto que casos de uso como *"Consultar lista de livros requisitados"* ou *"Consultar total de livros existentes"* são instanciações de casos de uso de interrogação.

O primeiro tipo de casos de uso é muito mais complexo, e da sua análise resultam inúmeras contribuições para as diversas fases da modelação e da construção do sistema. Um caso de uso que implique a alteração do estado interno do sistema está sujeito ao teste das condições em que tais alterações são possíveis, à especificação e validação sobre o que acontece quando termina e à própria descrição de quais são os passos necessários para que as alterações surtam efeito. Nessa perspectiva a análise destes casos de uso é determinante porque permite identificar os objectos envolvidos na interacção que são de três tipos: objectos que asseguram a comunicação com o utilizador (existentes a nível da camada de apresentação), objectos que guardam o estado interno do sistema (responsáveis pelas agregações e composições de outros objectos) e ainda objectos que implementam os algoritmos que possibilitam a pesquisa e transformação de informação.

Um caso de uso do tipo (i) é constituído por diversas interacções entre o utilizador e o sistema que apresentam uma padrão de diálogo identificado por Cockburn [Cockburn 01] a partir da definição inicial de Jacobson [Jacobson 92]. Este padrão é relevante porque são os casos de uso que são constituídos por diálogos destes que permitem que mais tarde se descrevam as expressões rigorosas que os definem.

A Figura 4.7, apresenta o comportamento que se obtém numa interacção típica.

Com este padrão de comportamento, é possível afirmar que um caso de uso pode ser visto como uma sequência de transacções, cada uma delas com as seguintes etapas:

1. o actor invoca a funcionalidade e envia os dados necessários;

2. o sistema valida o pedido e os dados fornecidos;

3. o sistema executa o algoritmo que suporta a funcionalidade e altera o seu estado interno, e

4. o sistema informa o actor do resultado da invocação.

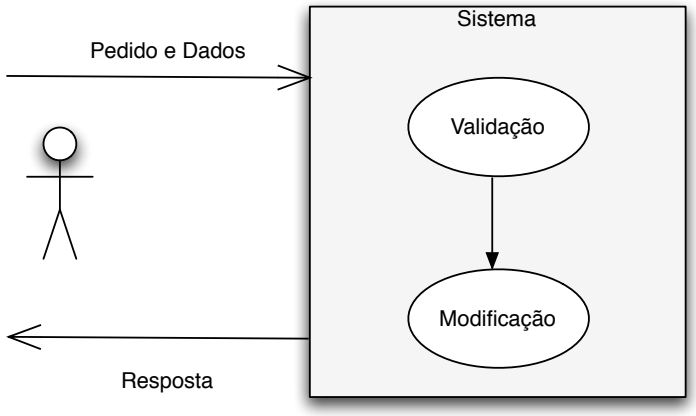


Figura 4.7: Padrão de um passo de um caso de uso.

No caso dos passos do caso de uso do tipo (ii) o passo 3 do padrão não se aplica na totalidade, visto que não se procede à alteração do estado interno[4].

Esta definição, permite que o caso de uso possa ser constituído por outros casos de uso que são incluídos na sua descrição. Do padrão identificam-se as diferentes responsabilidades que têm de ser previstas na concepção e construção dos sistemas para responder ao caso de uso. Do primeiro passo deriva-se a necessidade de construir objectos da camada de apresentação e que são responsáveis pela condução do diálogo interactivo com o utilizador. Não é líquido que o actor tenha de fornecer todos os dados de uma só vez ao sistema para este os validar. É possível, e desejável, que exista uma interacção entre o sistema e o utilizador na qual são trocadas as informações necessárias à prossecução do caso de uso. É ainda necessário contemplar a existência de entidades que assegurem a comunicação ao actor do resultado da invocação do passo do caso de uso (passo 4 do padrão).

Em relação ao terceiro item do padrão, existe a necessidade de providenciar o sistema de objectos que assegurem a representação do estado interno e objectos que assegurem as transformações que produzem os resultados. Esses objectos pertencem à camada de negócio e são internos ao sistema que se está a construir. Na aplicação do processo, se os objectos ainda não existirem (não tiverem sido identificados), é imperioso acrescentá-los ao modelo de domínio visto serem necessários à descrição da lógica de negócio.

O padrão apresentado, além de definir em que consistem as peças de um caso de

[4]Por vezes, e por uma questão relativa a factores ligados ao desenvolvimento, pode existir alteração de variáveis globais ao sistema.

uso, permite discorrer sobre a necessidade de especificação do mesmo, no que concerne às transacções que devem ser efectuadas, principalmente nos passos 2 e 3, de forma a garantir que a descrição da funcionalidade está correcta e completa e é bem desenvolvida posteriormente. O esforço necessário para a descrição de casos de uso dependentes do estado do sistema é, desta forma, substancialmente maior do que o empregue na descrição de casos de uso que não afectam o cálculo do valor do estado.

Estes casos de uso correspondem a sequências de transacções, sendo que se algo falhar no passo 2 de cada transacção o sistema não é modificado e o actor é informado. A especificação de comportamento destes casos de uso representa uma unidade de modularidade na forma como o processo os descreve. É sempre possível representar os casos de uso como possuindo uma descrição de comportamento que é facilmente descrita por uma sequência de interacções padrão, como se verá em secção posterior.

4.3.2 A importância do modelo de domínio

O processo apresentado pretende ser um auxiliar do trabalho do analista na tarefa de captura e análise dos requisitos de um sistema software. Não se pretende alargar o espectro de actuação a todas as fases do projecto de software, mas seria difícil não abordar uma parte determinante no processo de software: a definição do modelo de domínio.

Numa fase inicial do projecto de software o analista necessita de se identificar com o domínio envolvido para conseguir capturar de forma eficiente e não ambígua os requisitos envolvidos. A correcta percepção do modelo de domínio implica que o analista se dote das ferramentas necessárias para correctamente identificar as entidades relevantes do sistema, as regras de negócio envolvidas (os algoritmos) e as restrições existentes. O conhecimento do modelo de domínio permite ao analista estabelecer em concreto qual é o sistema que vai ser construído (ou reformulado) através do conhecimento de quais são as peças essenciais do mesmo. Tenha-se em atenção que o entendimento das regras de negócio e principalmente das restrições que essas mesmas regras definem é uma fonte de conhecimento muito importante, visto que desta forma se determinam as entidades do modelo, os seus relacionamentos, as restrições a esses mesmos relacionamentos e as restrições definidas em função do estado interno do sistema.

Este modelo, que após sucessivas etapas de iteração e refinamento, dará origem ao diagrama de classes do sistema, é importante que seja do conhecimento da equipa de projecto e do cliente. Não é declaradamente orientado à construção do sistema, o que significa que algumas das entidades representadas desaparecerão no diagrama de

classes, enquanto que outras surgirão, seja por necessidades do modelo de programação seja por informação retirada dos requisitos não funcionais.

Numa perspectiva de introdução de maior completude ao processo de captura e especificação dos requisitos do sistema, podemos dizer que este assenta em três vertentes fundamentais:

- modelo de (entidades de) domínio - permite ao analista identificar as entidades do domínio que estão envolvidos nas interacções que permitem a implementação dos casos de uso. A definição das entidades do modelo de domínio permite que se iniciem as actividades que conduzem à descoberta do controlo de fluxo associado às tarefas do sistema;
- modelo dos casos de uso - onde se identificam as funcionalidades do sistema como são visíveis do exterior e se determinam os actores a que estão associadas, e
- modelo de arquitectura do sistema - sendo que nesta fase a noção de arquitectura é bastante abstracta concretizando-se pela definição dos pacotes principais do sistema e das entidades do domínio da aplicação.

A especificação do modelo de domínio permite ao analista a agregação de diversa informação que é importante para as tarefas de modelação. Os objectivos principais da descrição do modelo de domínio são:

- organizar o vocabulário do domínio do problema que é utilizado na descrição dos casos de uso e diagramas subsequentes (por exemplo, nos diagramas de estado);
- capturar os requisitos da informação que é trocada entre o sistema e o exterior, ou que é mantida no sistema, e
- especificação das transacções do negócio, isto é, numa primeira fase os casos de uso e posteriormente, após um processo de refinamento, as operações e métodos que são suportadas.

Esta análise dá origem a que apareçam na modelação três tipos diferentes de entidades, que se concretizarão como classes no domínio da solução software. Esses três tipos de classes são:

- classes que modelam o estado interno e partilhado do sistema;

- classes que modelam a estrutura dos documentos recebidos e enviados (os *inputs* e *outputs* do sistema), e

- classes que modelam os tipos de dados que são trocados nas colaborações que permitem a concretização dos casos de uso.

A descrição do modelo de domínio é uma peça importante na análise de qualquer sistema software e qualquer processo de análise que não o tenha como auxiliar, arrisca-se a estar a laborar em bases pouco sustentadas. A ligação que se pode traçar a partir da descrição do modelo de domínio é bem evidente nas peças de análise que se seguem no processo. Tome-se o exemplo dos diagramas de caso de uso e correspondente descrição. Existindo uma definição concreta do modelo de domínio, as entidades do mundo do problema estão identificadas e nomeadas, sendo desejável que nos casos de uso se refiram *por nome* esses conceitos. Dessa forma, estabelece-se uma normalização sintáctica e define-se um mapeamento único entre a sintaxe e a semântica, para cada elemento do modelo de domínio.

4.3.3 Diagramas de Estados

Os diagramas de estado são uma das ferramentas utilizadas para a modelação da componente dinâmica do sistema. Os diagramas de estado, dado serem construídos numa óptica centrada numa entidade, possuem todos os estados, transições, eventos e actividades que são importantes no ciclo de vida dessa entidade.

Nas situações em que o ciclo de vida de uma entidade seja complexo, pode tornar-se de difícil análise e leitura a especificação descrita no diagrama de estados. Uma forma expedita que o analista possui para simplificar a complexidade de um diagrama de estados consiste na introdução de super-estados e a correspondente incorporação de decomposição hierárquica na elaboração do diagrama. Como resultado desta abordagem um super-estado num determinado nível é decomposto por diversos sub-estados num diagrama de estados mais específico.

A vantagem da utilização desta decomposição hierárquica é que tal permite que o analista quando pretende explorar apenas um componente do sistema não necessite de considerar todos os aspectos comportamentais do mesmo. Por exemplo, quando se pretende analisar apenas um cenário do caso de uso (seja o principal ou um alternativo) não é necessária a totalidade do diagrama de estados, mas apenas os estados, transições e eventos importantes para o cenário em causa. Nessas situações é possível abstrair

os restantes elementos do diagrama e criar os modelos de análise comportamental, tipicamente diagramas de interacção, apenas para o que se pretende estudar.

Note-se que qualquer diagrama de estados que faça uso de decomposição hierárquica pode ser transposto para um diagrama equivalente sem decomposição hierárquica, pelo que se pode afirmar que por cada diagrama de estados hierárquico existe um outro, sem esta estruturação, que lhe é semanticamente equivalente.

A Figura 4.8 apresenta um diagrama de estado para um aparelho de ar-condicionado, em que se utiliza o mecanismo de decomposição hierárquica. O diagrama poderia ser simplificado escondendo o detalhe do estado Em Aquecimento, se não se pretendesse detalhar a máquina de estados que este contém. De igual forma seria possível colocar todos os estados ao mesmo nível com o cuidado inerente à nova topologia de ligações que seria necessário descrever.

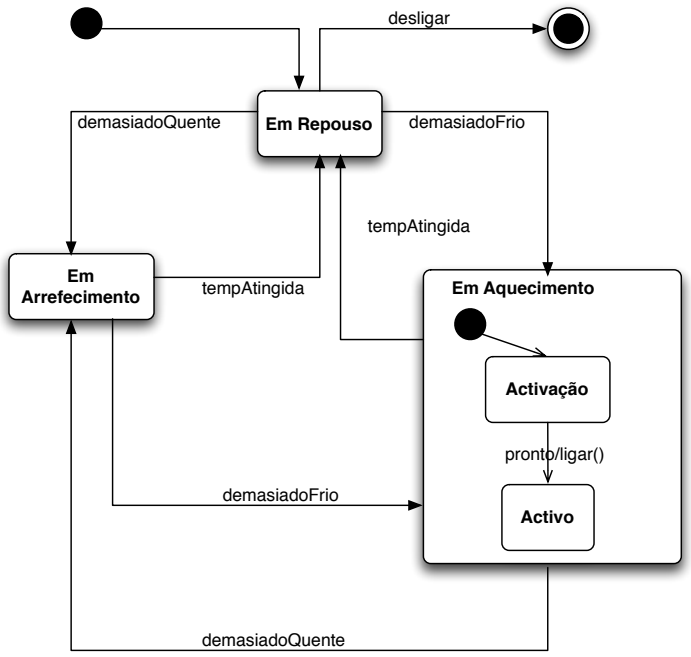


Figura 4.8: Diagrama de Estado com decomposição hierárquica.

A simplificação que a decomposição hierárquica oferece é visível em dois aspectos essenciais da construção do diagrama de estados: a criação de super-estados como elemento agregador de outros estados e a agregação de transições entre estados. Cada um destes aspectos resulta da simplificação feita aos diagramas originais, permitindo esconder e revelar (quando necessário) detalhes da modelação. A agregação de transições ocorre quando ao criar sub-estados se agrupa numa única transição as transições que

emanavam dos sub-estados, o que origina uma redução das transições que estão sujeitas a análise. No caso do diagrama para o ar-condicionado, se não existisse o estado Em Aquecimento, os estados Activação e Activo teriam de possuir, cada um deles, as transições de entrada e saída que de momento estão apenas ligadas ao super-estado.

Além da decomposição hierárquica, uma outra forma de simplificação de diagramas de estado complexos recorre à descrição de diagramas de estado com incorporação de concorrência interna. Esta forma de simplificação dos diagramas é por vezes descrita como sendo uma decomposição conjuntiva, visto que a descrição do comportamento se faz pela conjunção das descrições que são colocadas em concorrência. Tenha-se em consideração que apesar de se referir que os vários sub-diagramas de estado são colocados em concorrência esta construção pode ser utilizada para especificar diferentes aspectos comportamentais de uma entidade, desde que entre esses aspectos não exista interdependência.

O diagrama de estados da Figura 4.9 representa um diagrama de estados com concorrência interna, na medida em que o estado possui duas máquinas de estado internas que executam independentemente uma da outra.

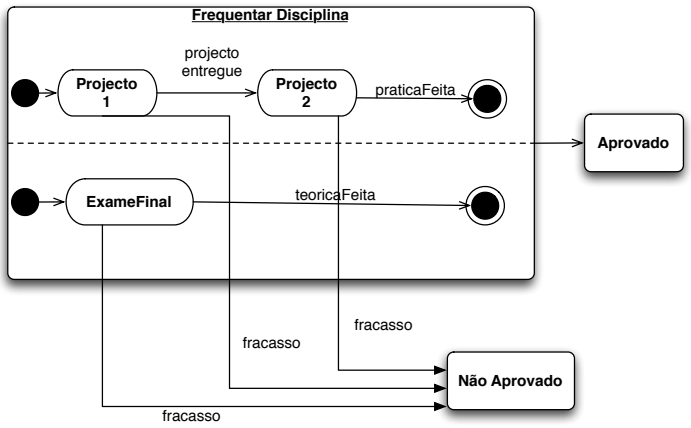


Figura 4.9: Diagrama de Estado com concorrência interna.

Com o recurso a estes mecanismos de simplificação torna-se mais evidente o processo de utilização dos diagramas de estado para a especificação comportamental da execução dos casos de uso. Não é necessário considerar em todos os momentos do processo de análise o diagrama com toda a sua complexidade, podendo o analista (eventualmente em conjunto com os utilizadores) escolher o grau de abstracção a que pretende trabalhar.

Construção dos diagramas de estado a partir dos casos de uso

Depois de descritos os casos de uso, usando quer a linguagem diagramática (fornecida pelo diagrama) quer o *template* de descrição de narrativa, é necessário acrescentar ao modelo as componentes comportamentais, que especificam o que acontece ao sistema e aos actores envolvidos durante a execução de um caso de uso. Esta descrição é feita com recurso à construção dos diagramas de estado, onde se explicitam os estados pelos quais as entidades (e o sistema) passam e as transições e actividades associadas.

Como mecanismo complementar ao diagrama de estados é necessário considerar os diagramas de interacção associados (os diagramas de sequência) que permitem estabelecer um outro domínio semântico ao possibilitarem a identificação das primeiras concretizações em termos de concepção e arquitectura de software. É nesta fase que se descobrem os objectos (numa primeira fase de negócio) que são imprescindíveis para que um determinado cenário de um caso de uso possa ser invocado.

O processo de construção de um diagrama de estado a partir de um caso de uso começa com a identificação de um cenário contido nesse caso de uso, ou seja, um caminho - um traço - na execução do mesmo. Geralmente o diagrama de estados começa a ser elaborado a partir do cenário normal, que apresenta a sequência de interacções mais comum entre os actores e o sistema. Para tal identificam-se as sequências de eventos exteriores que induzem a transição de um estado para outro, assim como também se faz o levantamento das actividades associadas à permanência num determinado estado do diagrama e às acções que eventualmente são desencadeadas pelo efectuar da transição. As actividades associadas a um estado são descritas de forma a se especificar se são feitas na entrada ou saída do estado, ou durante a permanência no mesmo. As acções e as actividades são determinadas pela análise que se faz das respostas que o sistema fornece dado um evento externo (que é normalmente despoletado pelo actor) e que estão presentes na narrativa do caso de uso.

Tendo em conta as facilidades de estruturação e gestão do nível de abstracção que se pode ter na construção de um diagrama de estados, inicialmente é criado um diagrama mais genérico que vai ser refinado levando a que se possam criar níveis de decomposição hierárquica para lidar com a complexidade. Os estados que fazem parte do diagrama devem ser todos visíveis externamente, isto é, do ponto de vista dos utilizadores do sistema, a identificação dos estados deve ser facilmente extrapolada à luz do domínio da aplicação. Tais estados representam consequências de acções desencadeadas pelos actores, de forma explícita ou implícita.

Depois de construído o diagrama de estados que descreve o cenário principal é ne-

cessário determinar todos os possíveis eventos externos que afectam o diagrama, ou seja, é necessário que o diagrama incorpore a descrição dos vários cenários alternativos. Para cada um desses cenários é necessário complementar o diagrama com os novos estados que se revelem necessários e com a especificação das acções e actividades envolvidas.

A título de exemplo, considere-se o exemplo clássico em que existe um caso de uso que possibilita o levantamento de dinheiro numa máquina ATM.

A descrição textual do caso de uso, refere que:

Caso de Uso	Levantar Dinheiro
Sumário	O cliente requisita o levantamento de determinada quantia de dinheiro da sua conta bancária
Actor	Cliente
Descrição	1. O cliente insere o cartão 2. O cliente insere o PIN do cartão 3. O cliente escolhe a operação "Levantar Dinheiro", introduz o montante desejado e confirma a operação 4. O sistema verifica se o cliente tem dinheiro suficiente na sua conta e se o limite diário de levantamento não foi excedido 5. Se tal for verdade, o sistema autoriza o levantamento 6. A máquina disponibiliza o dinheiro ao utilizador 7. A máquina ejecta o cartão 8. A máquina volta ao estado inicial e exibe a janela inicial
Fluxos Alternativos	1. Se o sistema determinar que o PIN é inválido informa o utilizador e ejecta o cartão 4a. Se o sistema determinar que o saldo actual da conta não é suficiente para o levantamento pretendido, informa o utilizador e ejecta o cartão 4b. Se o sistema determinar que o montante de levantamento diário foi excedido, informa o utilizador e ejecta o cartão

Este comportamento pode ser descrito através da criação de um diagrama de estado, onde se identificam as diferentes situações em que o caso de uso se pode encontrar e se definem as acções que são tomadas em cada uma delas. As situações em que um caso de uso pode estar correspondem aos estados do diagrama e as acções correspondem às

transições.

A Figura 4.10 ilustra o diagrama que se pode construir para descrever o comportamento do caso de uso.

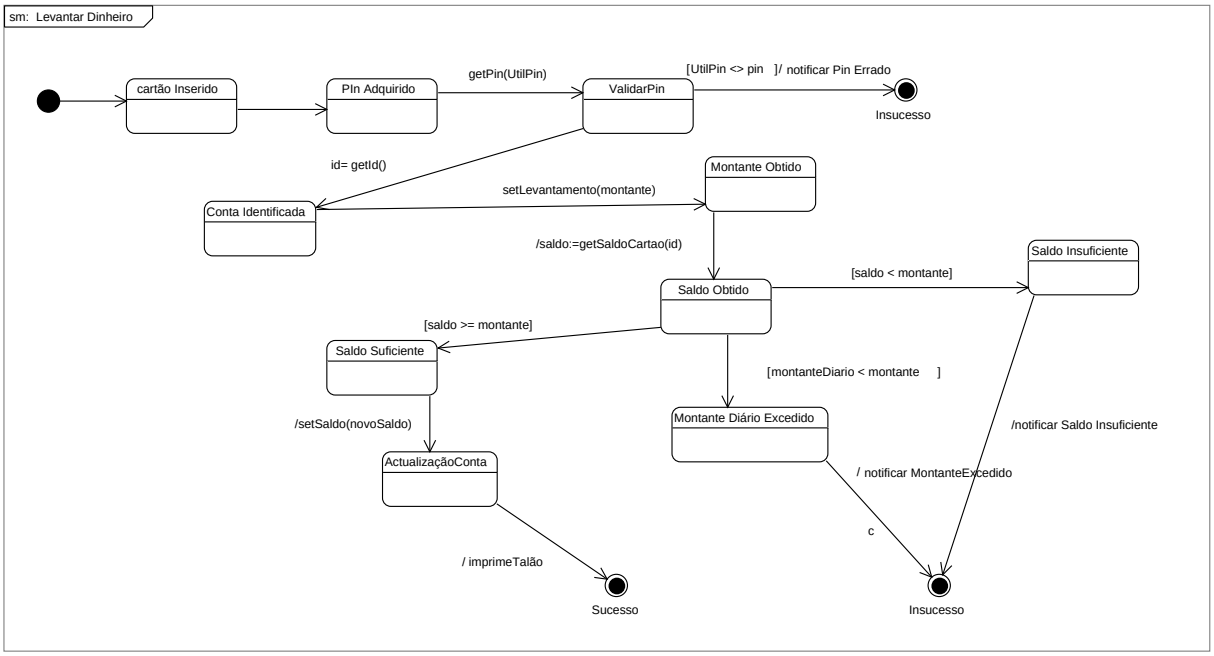


Figura 4.10: Diagrama de estados para "Levantar Dinheiro".

Os estados correspondem às situações em que a descrição determina que o caso de uso pode estar e as transições asseguram o fluxo de controlo. No caso do diagrama de estados actual, optou-se por não colocar nas transições muito detalhe, servindo o exemplo apenas para ilustrar o processo.

Modelação da componente dinâmica

A modelação da componente dinâmica de um sistema é baseada numa estratégia *use-case driven*, na medida em que para cada caso de uso se determina a forma como é

que a invocação deste é concretizada, ou seja, como é que as entidades do sistema software interagem entre si com o objectivo de satisfazer os objectivos do caso de uso. Para cada caso de uso é necessário determinar qual o controlo de fluxo que o rege e posteriormente quais os objectos do sistema que estão envolvidos na sua prossecução.

A Figura 4.11 apresenta graficamente o processo de modelação orientado ao caso de uso (*use-case driven*), que é utilizado pelo modelo de processo deste trabalho.

O processo inicia-se no caso de uso que é especificado passo a passo, descrevendo com detalhe a forma como a interacção entre o actor e o sistema é conduzida. Nessa descrição utilizam-se os conceitos do modelo de domínio, visto serem os únicos que são no momento conhecidos pela equipa de projecto. Para cada caso de uso e de acordo com a sua narrativa textual produzem-se os modelos comportamentais que o descrevem, isto é, criam-se os diagramas de estado e de sequência que especificam respectivamente o seu ciclo de vida e as interacções existentes.

A análise da componente dinâmica de um caso de uso começa com a identificação do estímulo externo, produzido do actor que desencadeia o caso de uso e que usualmente é efectuado sobre um objecto que pertence à interface do sistema. Posto isto, dá-se origem a uma sequência de acções e trocas de mensagens entre o sistema (e seus componentes internos) e o actor que invocou o caso de uso.

Na elaboração das descrições comportamentais, identificam-se os diferentes objectos envolvidos no diálogo e a forma como essa interacção é desenvolvida. A interacção entre os objectos deve ser descrita num diagrama de interacção e esse diagrama reflecte as trocas de mensagens e os eventos que são descritos no diagrama de estados. O diagrama de estados que especifica o comportamento do caso de uso, como sendo uma entidade do sistema, contém todas as acções, actividades e estados que descreve o ciclo de vida de uma invocação do caso de uso. O diagrama de sequência pode representar apenas um cenário do caso de uso, logo pode corresponder apenas a um caminho que se identifica no diagrama de estados.

Os diagramas de sequência que se criam durante a fase de análise apresentam um nível de detalhe e de refinamento que não é comparável aos mesmos diagramas, em fases mais adiantadas do projecto, como sejam as fases de concepção e desenvolvimento. Na fase de análise ainda não existe uma percepção totalmente clara e bem definida de quais serão as entidades da arquitectura de classes, mas o analista trabalha ao nível do domínio de negócio, conhecendo e utilizando as entidades de negócio que estavam evidentes nos requisitos. Algumas destas entidades de negócio constituir-se-ão como entidades do domínio do sistema software, enquanto que outras através do processo de refinamento serão transformadas e dão origem a novas entidades.

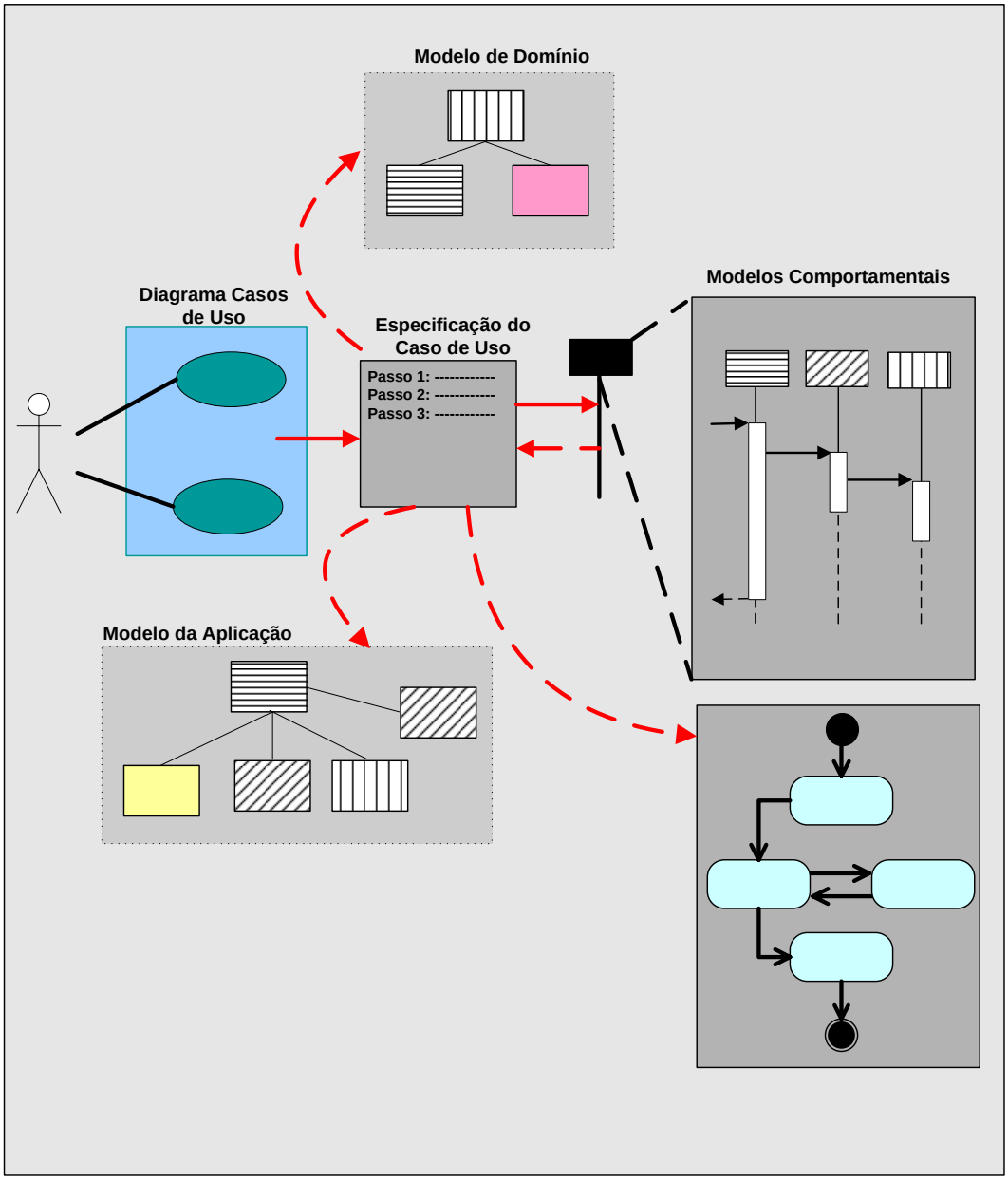


Figura 4.11: Processo de modelação *Use Case Driven*.

Note-se que na imagem, os diagramas de sequência possuem como elementos da interacção entidades que são do modelo de domínio, os objectos de negócio, mas também estão representadas entidades que foram entretanto descobertas e que fazem também parte da arquitectura da solução. Os objectos graficamente representados com fundo de textura com traços horizontais e verticais são do modelo de domínio, enquanto que o objecto com fundo de textura com traços diagonais que está envolvido na interacção pertence apenas ao modelo da aplicação, tendo sido descoberto durante a construção dos modelos comportamentais do caso de uso.

Um outro aspecto que merece registo tem a ver com a falta de clareza, nesta fase, na definição da actividade dos objectos envolvidos e se eles são activos ou não. Essa definição é feita apenas na fase de concepção aquando da criação da arquitectura do sistema e da definição de como é que os algoritmos são transformados para o modelo de programação escolhido.

Comportamento dependente do estado

Na análise aos casos de uso de um sistema, uma divisão pode ser traçada entre aqueles casos de uso que são manifestamente dependentes do estado e cujo ciclo de vida é feito em função dos valores do estado e entre aqueles cujo comportamento não depende da história e portanto, não possuem qualquer noção de estado. Exemplo destes últimos são os casos de uso que independentemente do estado do sistema apresentam sempre o mesmo resultado.

Mais interessantes do ponto de vista da construção de sistemas software são os casos de uso que possuem um comportamento que podemos designar de reactivo, em que o resultado apresentado corresponde à forma como o sistema reage em função dos valores actuais do seu estado interno e da sua história. Nestas situações, o caso de uso deve ser descrito por um modelo que esteja definido com base num meta-modelo baseado em estados, o que na domínio da linguagem UML significa que o formalismo adequado para o fazer é o diagrama de estados.

Um caso de uso que seja dependente do estado é um caso de uso em que os resultados apresentados pelo sistema não dependem apenas das entradas fornecidas pelo actor, mas também pelo que anteriormente aconteceu ao sistema. Este tipo de casos de uso deve possuir um objecto que funciona como controlador do estado e cujo ciclo de vida é definido por um diagrama de estado que especifica o controlo de fluxo e a sequência de execução do caso de uso. Tenha-se em atenção que o diagrama de estados pode não especificar a funcionalidade de uma operação, mas todo o comportamento do caso de

uso, o que potencialmente pode envolver o recurso a várias operações do sistema.

Na análise de casos de uso com dependência de estado o objectivo do analista passa por determinar a interacção entre as seguintes entidades:

- o objecto que executa o diagrama de estados, ou seja, o objecto com o controlo de fluxo dependente do estado;
- os objectos que enviam eventos para o objecto de controlo e que provocam as transições. Estes objectos são usualmente objectos da interface com o utilizador;
- os objectos que são responsáveis pela execução das acções desencadeadas nas transições e pelas actividades efectuadas durante a permanência num determinado estado, e
- outros objectos que sejam relevantes para a prossecução do caso de uso.

A forma como estes objectos colaboram entre si estará explícita num diagrama de interacção. As mensagens presentes no diagrama de sequência são constituídas pelo eventos e pelos dados transportados na transição. Quando uma mensagem é enviada ao objecto de controlo isso corresponde a uma transição no diagrama de estado correspondente, sendo que por simplificação apenas nos interessa a componente *"evento"* da mensagem.

A acção no diagrama de estado é o resultado da transição para um novo estado e corresponde no diagrama de sequência ao evento gerado como resultado do envio da mensagem anterior (se esta provocar a ocorrência de resposta).

A chegada de um evento causa uma transição no diagrama de estados que desencadeia, isto é, tem como efeito, um ou mais eventos de resposta (ou saída). Estes eventos correspondem à reacção do objecto de controlo e são concretizados no diagrama de estados como sendo acções associadas quer à transição de estado, quer à entrada ou saída de um estado. O algoritmo base que define a análise dinâmica de um cenário dependente do estado em sistemas que são na sua essência maioritariamente reactivos, pode ser resumido nas seguintes acções:

- determinar os objectos da camada interactiva - são os objectos que recebem os eventos produzidos pelo actor na sua interacção com o sistema;
- determinar o objecto de controlo - este é o objecto que executa o diagrama de estados e que corresponde à entidade que implementa o algoritmo que mapeia a funcionalidade associada ao caso de uso;

- determinar outros objectos internos - correspondem aos objectos envolvidos na interacção e que trocam mensagens com o objecto de controlo ou com os objectos da interface. Usualmente estes objectos existem na camada de negócio e são eles os responsáveis pela implementação das regras de negócio;

- determinar a execução do diagrama de estado - permite especificar qual é o estado do objecto de controlo, o seu ciclo de vida, tendo em conta os eventos que são desencadeados pela interacção dos diferentes objectos envolvidos no diálogo;

- determinar interacções (colaborações) entre os objectos - esta fase é feita em conjugação com a anterior, de forma a descrever com detalhe a interacção entre o objecto de controlo e o diagrama de estado que ele executa, e

- determinar execução dos cenários alternativos - depois de construído o diagrama de estado que o objecto de controlo executa, é necessário anotá-lo com os estados e transições necessários para descrever os cenários alternativos do caso de uso.

A identificação destes objectos, concretizada durante a aplicação do processo, possibilita que a equipa de projecto recolha informação que iterativamente enriquece o modelo da aplicação, através da descoberta de objectos que serão parte da solução.

4.3.4 Diagramas de Sequência

As descrições dos casos de uso permitem especificar as interacções entre o actor e o sistema aquando da invocação de uma funcionalidade no sistema. O caso de uso é detalhado através da identificação dos diversos cenários em que se materializa, isto é, as diferentes alternativas que a sua execução permite. Cada cenário que se pode retirar da descrição corresponde a um caminho no diagrama de estados.

A descrição do caso de uso corresponde a uma estrutura de controlo condicional, na qual existem vários cenários, desde o cenário normal (o cenário positivo) e uma série de cenários alternativos que correspondem às execuções que ocorrem quando determinadas condições são verdadeiras. É normal, que na maioria dos sistemas, estes cenários alternativos correspondam à detecção de situações de excepção. Caso não fosse esta a razão, provavelmente o analista deveria ter dividido o caso de uso em casos de uso mais elementares. A representação destes cenários alternativos é feita na descrição textual e também no diagrama de casos de uso através da utilização de $<< extend >>$.

Cada um dos cenários, correspondente a um caminho no diagrama de estado, dá origem a um diagrama de sequência que especifica a troca de mensagens que ocorre

ao nível dos objectos envolvidos. Com a capacidade de composição que os diagramas de sequência permitem com o recurso às *frames* de interacção, é possível descrever no mesmo diagrama a narrativa completa do caso de uso, utilizando a capacidade de modularidade oferecida pela notação.

```
if (pre_condição) then
  //cenário normal
else
 if (condição_1) then
  //cenário alternativo 1
 if (condição_2) then
  //cenário_alternativo 2
 if (...) then
  // ...
```

Figura 4.12: Algoritmo genérico de um caso de uso.

A Figura 4.12 corresponde, em linguagem algorítmica, ao controlo de fluxo de um caso de uso, com o seu cenário normal e os diferentes cenários alternativos. O diagrama de sequência que é visível na Figura 4.13 representa a forma como pode ser descrito o caso de uso. O diagrama de sequência apresentado constituí um padrão genérico para a descrição do comportamento de um caso de uso. Por uma questão de simplificação e facilidade de construção, o analista pode separar cada um dos blocos em diagramas independentes.

Tal como se fez para os diagramas de estado, apresenta-se na Figura 4.14 o diagrama de sequência que descreve o cenário principal do caso de uso "Levantar Dinheiro". Tal como no correspondente diagrama de estado, esta é uma versão simplificada e que não apresenta demasiado detalhe acerca das trocas de mensagens. Ilustra apenas uma interacção básica entre o actor e o sistema, mas introduz na sua descrição entidades que não tinham sido necessárias no diagrama de estados, mas que o diagrama de sequência tem de instanciar. Essas entidades devem constar do modelo de domínio ou, caso contrário, são entidades que foram descobertas e que pertencerão ao diagrama de classes do modelo da aplicação.

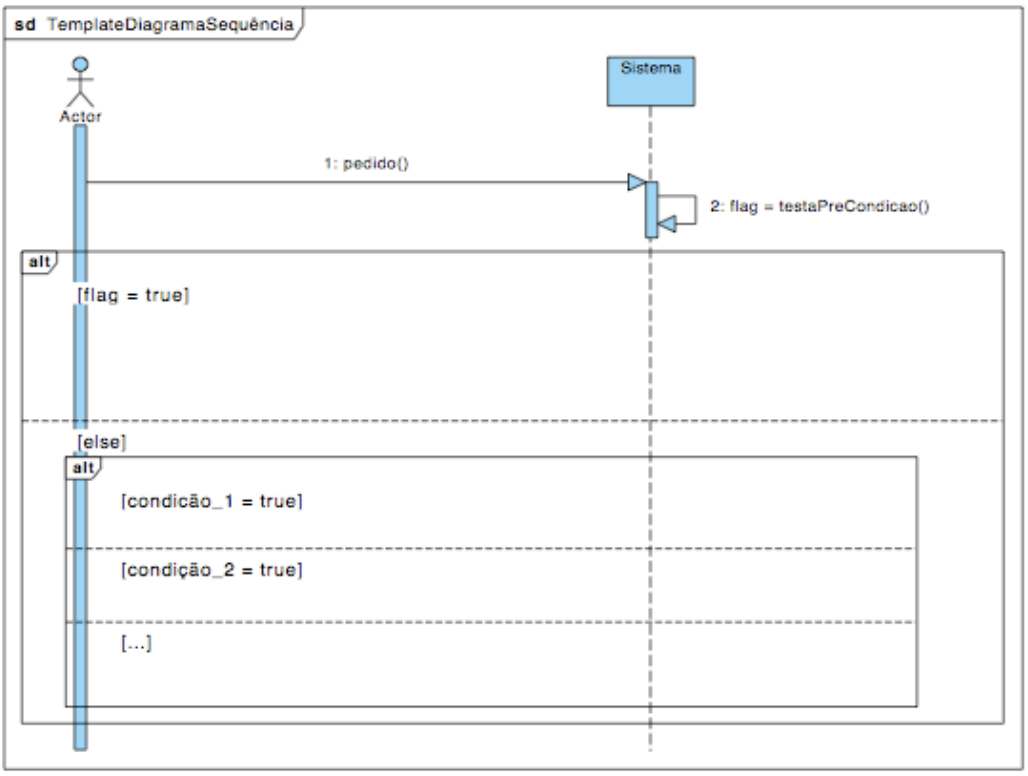


Figura 4.13: Estrutura padrão de Diagrama de Sequência.

4.4 Expressões rigorosas no diagrama de casos de uso

A identificação da funcionalidade visível, ou percebida, de um sistema e a caracterização dos actores que intervêm no processo é o objectivo desta fase da análise. Os casos de uso permitem descrever de forma informal o sistema, usualmente recorrendo à utilização de texto em língua natural. É natural que desta abrangência na definição surjam problemas na escolha do nível de abstracção desejado, identificáveis a dois níveis:

1. *generalização em demasia*: não permitindo que seja transmitida uma clara descrição de qual é o comportamento em causa. Exemplos desta generalização em demasia são casos de uso de complexidade relevante e apenas descritos a nível do diagrama de casos de uso como "Requisitar Livro" num sistema de gestão de uma biblioteca, "Iniciar Produção" numa sistema de controlo de uma central termoeléctrica, entre outros. Um elemento da equipa de projecto não pode inferir

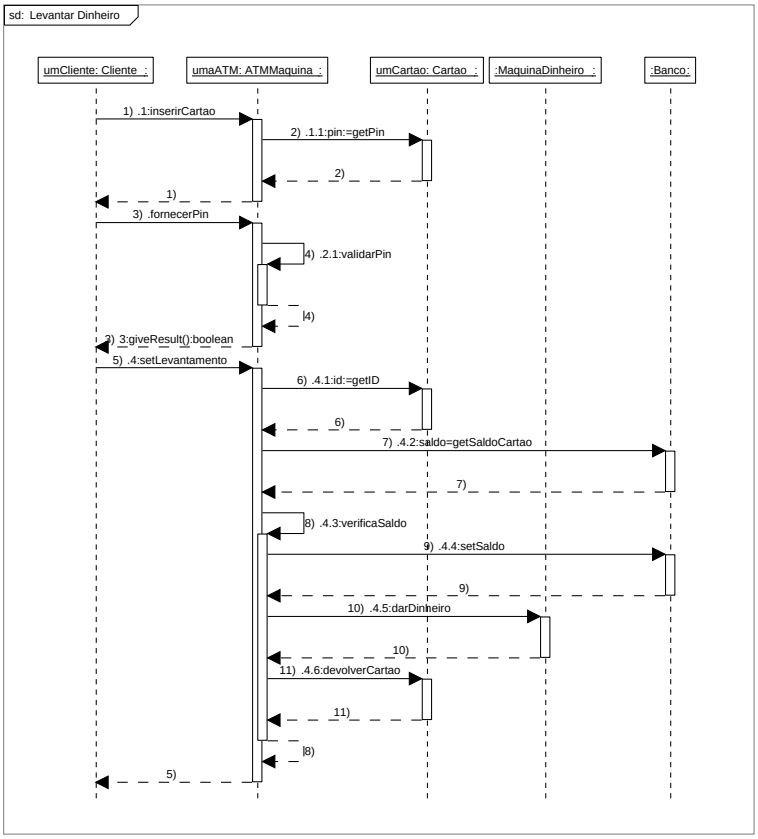


Figura 4.14: Diagrama de sequência de "Levantar Dinheiro".

o comportamento a partir desta descrição e não pode assumir que existem outras peças de documentação onde a especificação esteja descrita se esta não estiver relacionada com o diagrama de casos de uso.

2. *detalhe em demasia*: permitindo que o diagrama de casos de uso fique muito exposto a detalhes pouco relevantes e demasiado concretos, para esta fase. Esta é uma situação típica de quando o processo tem muitas variantes e no diagrama de casos de uso existe a tendência para colocar todas as situações, quer elas constituam de facto funcionalidade relevante, ou apenas auxiliares, ao nível da operação. Um exemplo típico deste excesso de verbosidade visual na identificação dos casos de uso ocorre na descrição da camada interactiva na tentativa de descrição do diálogo. Por vezes este excesso de detalhe é também induzido pela tentativa de se pretender descrever fluxos de controlo nos diagramas.

A natureza informal dos casos de uso permite que uma descrição mais ou menos elaborada possa ser considerada como um caso de uso. Esta característica indesejável causa muitos problemas devido à informalidade que introduz, não permitindo que se escolha o nível correcto de abstracção para descrever casos de uso. A identificação e descrição de um caso de uso está longe de ser uma tarefa trivial, como é amplamente discutido na literatura [Constantine 01, Génova 02, Isoda 03, Armour 01].

Como se viu no Capítulo 3 existem alternativas que descartam esta forma de especificação e constroem modelos paralelos, noutras linguagens, fazendo com que o processo de análise se desenvolva a outro nível. No entanto, essa alternativa não permite que a modelação se faça recorrendo unicamente à linguagem UML.

O Rational Unified Process define um arquétipo (um *template*) de descrição de um caso de uso, como ferramenta complementar ao diagrama de casos de uso onde se pretende colocar alguma formalidade na descrição (pelo menos de ordem sintáctica). Muitas abordagens distintas das da Rational criaram o seu estilo próprio para a descrição dos casos de uso, acrescentando informação ao mesmo, mas o espírito que preside à sua descrição mantém-se.

Veja-se o exemplo clássico, retirado de [Gomaa 00], de uma descrição de caso de uso apresentado na Figura 4.15, e que tem sido utilizado ao longo do capítulo.

Nesta descrição existe já informação relevante, embora não suficientemente detalhada, sobre o comportamento do caso de uso e sobre as restrições à execução do mesmo. Falta no entanto, que esta informação seja estruturada de forma mais conveniente e que se perceba de forma mais clara as diversas interacções existentes. As restrições que estão expressas no caso de uso, são perceptíveis pelos utilizadores do sistema, sendo que são eles os interlocutores da equipa de projecto nesta fase. Nestas narrativas, estão explicitadas os quatro passos do padrão de um caso de uso, como explicitado em 4.7, e que correspondem à execução de uma funcionalidade.

Apesar de toda a estruturação existente e do nível de detalhe que se possa ainda adicionar à descrição feita, este tem de ser forçosamente definida como sendo informal.

O propósito deste trabalho consiste na introdução de rigor na utilização da linguagem, especialmente nos diagramas de caso de uso, sendo que este caminho da informalidade para a formalidade será frutífero se permitir que se introduza mais cedo no processo de análise aspectos que influenciarão as decisões que são relativas às fases de concepção e de desenvolvimento. Existe uma motivação clara, e bem explícita, para introduzir esta componente mais formal na fase de análise, visto que esta é a única fase em que o cliente e demais utilizadores podem dar o seu contributo. Esse

Caso de Uso	Levantar Dinheiro
Sumário	O cliente requisita o levantamento de determinada quantia de dinheiro da sua conta bancária
Actor	Cliente
Dependências	Caso de uso "Validar PIN" deve estar incluído
Pré-condição	A máquina ATM está em inactividade, exibindo a janela inicial
Descrição	1. Caso de uso "Validar PIN" é invocado 2. O cliente escolhe a operação "Levantar Dinheiro", introduz o montante desejado e confirma a operação 3. O sistema verifica se o cliente tem dinheiro suficiente na sua conta e se o limite diário de levantamento não foi excedido 4. Se tal for verdade, o sistema autoriza o levantamento 5. A máquina disponibiliza o dinheiro ao utilizador 6. A máquina imprime um recibo com a informação da transacção e com o saldo actual da conta 7. A máquina ejecta o cartão 8. A máquina volta ao estado inicial e exibe a janela inicial
Fluxos Alternativos	1. Se o sistema determinar que o PIN é inválido informa o utilizador e ejecta o cartão 3a. Se o sistema determinar que o saldo actual da conta não é suficiente para o levantamento pretendido, informa o utilizador e ejecta o cartão 3b. Se o sistema determinar que o montante de levantamento diário foi excedido, informa o utilizador e ejecta o cartão 5. Se a máquina ATM não dispõe de dinheiro suficiente para satisfazer o montante de levantamento, informa o utilizador, ejecta o cartão e exibe o ecrã que sinaliza que está sem dinheiro
Pós-condição	A conta do utilizador tem um novo valor para o saldo resultante do decréscimo induzido pelo levantamento

Figura 4.15: Descrição do caso de uso "Levantar Dinheiro"numa máquina ATM.

contributo é razão directa do tipo de técnicas utilizadas, pois os utilizadores ao perceberem a linguagem visual e ao efectuarem a descrição do caso de uso, conseguem também fornecer a informação relevante à sua concretização. Não seria possível que esta colaboração dos utilizadores fosse feita ao nível da concepção do sistema, na sua componente arquitectural, tal como é entendida e especificada ao nível dos diagramas de classe.

O cliente e os utilizadores, que são realmente os *donos* do sistema, devido ao seu conhecimento da área de aplicação, deverão ter a capacidade de fornecer à equipa de projecto a informação necessária para que o caso de uso seja devidamente definido, em todas as suas vertentes e cenários.

Dado não ser viável supor que o cliente seja capaz de expressar o conhecimento que possui de modo formal, através da utilização de métodos e modelos formais, é papel dos analistas a captura estruturada dos requisitos expressos e a sua transformação em modelos semanticamente mais bem fundados.

Deve ser possível pedir ao cliente para descrever os casos de uso explicitando também informação relevante sobre:

1. restrições à execução do caso de uso;

2. condições que devem ser verdadeiras quando o caso de uso é executado, e

3. condições a satisfazer pelo estado interno do sistema após a execução do caso de uso.

Às primeiras chamaremos de *pré-condições* e representam as condições necessárias para a execução correcta do caso de uso.

A *pós-condição* apresenta as condições que devem ser verdadeiras para que a execução de um caso de uso tenha sido bem sucedida[5].

O *invariante* descreve as condições que devem ser verdadeiras para que o estado interno do sistema (ou de cada entidade do sistema) seja coerente. O registo do invariante do sistema, escrito de forma clara é uma informação fundamental, dado que na maior parte dos casos pode não ser possível realizar a sua inferência completa a partir dos dados existente nos múltiplos diagramas.

[5]No caso de uso da Figura 4.15 a pós-condição expressa o facto que o saldo diminui, de determinado montante, após a conclusão do mesmo.

Casos de uso como contratos

Quando o cliente de um sistema que está a ser desenvolvido define os requisitos, a forma como o faz e o resultado desse processo, assume a forma de um *contrato*. O incremento semântico verificado na descrição de casos de uso com contratos, que são concretizados através da inclusão de pré e pós-condições nos mesmos [Cockburn 01, d'Souza 99], permitem um maior grau de formalização e disciplina no processo de recolha e validação dos requisitos.

A utilização da noção de contrato na descrição dos casos de uso deriva do *Design by Contract* de Bertrand Meyer [Meyer 92] e permite o enriquecimento da descrição dos requisitos. O conceito central do Design by Contract baseia-se na noção de *asserção*, que corresponde a uma expressão booleana que, para que o contrato seja válido nunca deverá ser falsa. Em [Meyer 92] identificam-se três tipos de asserções: as pré-condições, as pós-condições e os invariantes.

Tipicamente, as asserções são testadas de forma automática, se possível durante a fase de testes. Como os contratos especificam as condições críticas de funcionamento, é normal que estas sejam exaustivamente testadas, para que se possa verificar a conformidade do sistema com o que foi descrito e solicitado pelo cliente.

Com base neste incremento de rigor nas descrições dos casos de uso, apesar de estas ainda possuírem um nível de abstracção bastante elevado, é possível pensar em efectuar tarefas de validação e de verificação da correcção do seu comportamento.

A aplicação da noção de contrato aos casos de uso tem por objectivo a definição de restrições que permitem satisfazer as regras do negócio. A este nível de abstracção o que se define não é relativo às operações e artefactos de construção do sistema, mas relaciona conceitos e objectos do modelo de domínio. Os contratos expressos não garantem que o sistema vá funcionar correctamente caso o contrato não seja cumprido. A sua definição apenas explicita o que deve acontecer ao negócio se o contrato for cumprido.

É importante referir que do ponto de vista da modelação UML um contrato estabelece um domínio semântico que pode ser descrito no diagrama de classes. Por serem artefactos do modelo de domínio, os contratos possibilitam o estabelecimento de ligação entre os requisitos funcionais e as entidades relacionadas com eles. A informação extrapolada dos casos de uso, com as restrições que estes documentam, é importante para a correcta definição do diagrama de classes do modelo da aplicação, prevendo este que as associações entre as entidades devem estar sujeitas às regras que se podem extrair dos contratos.

4.4.1 Utilização da OCL

Não existe um suporte explícito na UML para a descrição de informação relativa às pré e pós-condições nos diagramas de caso de uso, mas a notação tem uma linguagem de escrita de restrições, a Object Constraint Language (OCL), que pode ser utilizada para tornar rigorosa a informação dos modelos UML e por consequência dos casos de uso. A OCL é uma linguagem formal, com construtores dotados de semântica bem definida e que é normalmente associada à fase de concepção, por permitir estabelecer restrições ao relacionamento entre classes, à componente algorítmica das operações no diagrama de classes e nas suas declinações no diagrama de estados e de sequência.

Com a proposta de utilização da OCL não se cria uma ruptura com as ferramentas oferecidas pela notação, uma vez que a OCL faz parte da plataforma UML, utilizando-se a linguagem de restrições noutras vistas que não as habituais. A utilização da OCL na construção do diagrama de casos de uso permite identificar a informação que pode ser descrita de forma rigorosa e que alimenta as fases seguintes do processo. Como efeito lateral deste processo, na passagem de informação informalmente descrita para informação rigorosa, regista-se uma maior precisão, visto que a passagem para modelos rigorosos faz com que a informação que estava implícita nas descrições textuais tenha forçosamente que se tornar explícita. Esta é uma mais-valia do processo, porque se mais benefícios não houvesse, existiria sempre o reforço concreto das tarefas de análise. A necessidade de com os utilizadores escrever as expressões que detalham aspectos dos casos de uso permite tornar as descrições textuais mais rigorosas, possibilitando também uma melhoria na documentação produzida.

É natural que algumas das expressões OCL levantadas nesta fase de análise sejam também importantes nos diagramas de classe, o que valoriza a abordagem, mas não é objectivo que todas as decorações feitas a esses diagramas sejam levantadas nesta fase de análise. Algumas das restrições escritas no diagrama de classes abordam aspectos de conceptualização da solução software que não são interessantes quando se discute a funcionalidade básica e as interacções existentes, como é feito no diagrama de casos de uso. Uma parte das restrições existentes no diagrama de classe pode derivar inclusive da transformação de modelos que conduz à elaboração de uma determinada solução arquitectural para o sistema, sendo que essas mesmas restrições podem ter sido colocadas para assegurar a correcta interpretação de associações entre classes.

O facto de não se prever que a OCL possa ser utilizada na descrição das restrições dos casos de uso, é também consequência da falta de definição de onde a linguagem deve ser utilizada e de um processo que descreva a integração da OCL com os restantes

diagramas. Existem duas razões para esta falta de definição. A primeira radica no facto de o espectro de utilizadores usuais da UML preferirem a utilização dos construtores visuais e da capacidade de abstracção que estes proporcionam e não terem de se preocupar com uma notação que é na essência semelhante às linguagens de programação. A segunda razão deriva do facto de uma vez que a UML não é uma metodologia, mas apenas uma notação, ser da responsabilidade da metodologia utilizada e respectivo processo, a definição de como é que a OCL é usada, em que fases e com que intuito.

A colocação de pré e pós-condições permite que os clientes forneçam informação sobre as restrições à execução de um caso de uso. Essa informação permite melhor identificar os possíveis cenários de um caso de uso e introduz rigor na especificação desses cenários.

A introdução de OCL na especificação de casos de uso permite congregar dois grandes propósitos:

1. manter a descrição informal dos casos de uso, permitindo um grau de liberdade que possibilite a comunicação facilitada com os utilizadores. Permite-se que a descrição se faça com o recurso ao vocabulário do utilizador e preserva-se a capacidade de o utilizador perceber as peças de documentação que são a descrição do caso de uso e o diagrama de casos de uso[6], e

2. adicionar informação que possibilite a descrição rigorosa das pré e pós-condições dos casos de uso. O utilizador tem esta informação implícita no conhecimento que possui do problema e ao explicitá-la fornece informação sobre o estado do sistema e sobre a capacidade de raciocinar sobre os fios de execução possíveis.

O objectivo do uso da OCL e da captura das pré e pós-condições na descrição do caso de uso permite que:

- se formalizem componentes da descrição dos casos de uso, permitindo o aproveitamento dessas expressões para subsequente prova, e

- se descubram restrições durante a fase de análise que são importantes para a fase de concepção. Acredita-se que a descoberta numa fase prévia permite antecipar a conceptualização da solução, possibilitando a identificação das situações de excepção e dos correspondentes mecanismos de controlo e recuperação.

[6]Por manifesta dificuldade de compreensão dos outros diagramas.

A captura destas restrições é particularmente relevante para a definição da camada de interface com o utilizador, visto que é usual que as restrições existentes ao nível da camada de negócio se reflictam primeiro ao nível do diálogo com o utilizador [Martins 95]. Entre essas restrições encontram-se todos os testes de pré-condições que determinam se as funcionalidades podem ser invocadas e que estando expressas na camada de negócio, por uma questão de correcta definição do controlo de diálogo com o utilizador, devem ser testadas o quanto antes e deve ser fornecido a quem invocou a funcionalidade o teste do resultado (*feedback*) sobre a possibilidade de execução da mesma. Por exemplo, num sistema de inscrições nos serviços académicos de uma escola, só deverá ser possível a um aluno inscrever-se numa determinada disciplina, se essa disciplina de facto existir. Faz todo o sentido testar a pré-condição, expressa na associação entre entidades do diagrama de classes, na camada interactiva como medida para melhorar a usabilidade da mesma. Não faria sentido recolher toda a informação passada pelo utilizador e esperar pela validação *total* efectuada ao nível da camada de negócio.

4.5 Casos de Uso e OCL

A invocação de um caso de uso corresponde a uma série que invocações que são feitas ao sistema. Essas invocações correspondem à funcionalidade que o actor escolhe e às funcionalidades auxiliares dessa, que são necessárias para a correcta prossecução do comportamento associado. No fim da invocação a pós-condição deve corresponder às mudanças induzidas pelos vários passos necessários para a prossecução do caso de uso. Por simplificação é necessário assumir que a pré-condição inicial do caso de uso é verdadeira, ou seja, o caso de uso pode ser invocado. Não interessa neste contexto prever o que pode acontecer se o caso de uso não for invocado, porque essa premissa não acrescenta nada ao raciocínio sobre o que acontece quando ele é invocado.

A validação da pré-condição é facilmente concretizada e a verificação de que foi efectuada é possível de ser acompanhada pela equipa de projecto e pelos utilizadores. A pós-condição assume contornos de maior complexidade porque o encadeamento de pós-condições (por exemplo na execução de uma tarefa) pode tornar o estado interno inconsistente, sendo necessário determinar quando é que tal aconteceu.

Quando o caso de uso é completamente executado, então nessa altura a pós-condição final terá de ser garantida pela pós-condição do último passo. Designemos por $Post_{UC}$ a pós-condição final do caso de uso e designemos a sequência de passos que modelam um caminho possível para a concretização do caso de uso por s. Todas as restrições a

serem escritas em OCL referem-se ao objecto s, donde a referência `self` será igual a s, isto é $self = s$.

Para determinar a sequência de acções e as alterações ao estado interno, é necessário que se detalhe o caso de uso e a sequência de acções do mesmo na forma de um diagrama de estados, de modo a se determinarem as várias alternativas de execução do caso de uso.

Seja Sp todas as sequências de acções (em sentido lato) possíveis de serem gerados pela execução do caso de uso. É possível criar um diagrama de estados que descreve todas as possibilidades de caminho que estão contidas em Sp. Cada caminho nesse diagrama de estados corresponde a uma sequência de acções que é possível identificar na execução do caso de uso. Este diagrama de estados regista os eventos que correspondem a operações que são invocadas pelo actor, que podem ser parametrizadas, e que desencadeiam transições de estado e, possivelmente, execução de acções.

A pós-condição do caso de uso, $Post_{UC}$, descrita de forma informal e é um elemento da fase de análise, enquanto que as condições que são colocadas no diagrama de estados assim como as pós-condições são já elementos que também podem ser da fase de concepção. As expressões em OCL são transportadas para essa fase e para os diagramas a ela associados, por serem expressões rigorosas que modelam regras e conceitos de negócio.

Num diagrama de estados podem encontrar-se todas as sequências p possíveis de traçar desde o estado inicial até um outro estado (que se considere como final), sendo que a cardinalidade desse conjunto depende da complexidade do diagrama de estados. Para cada sequência de caminhos p, com acções como $op_1(args1), ..., op_k(argsk)$, a operação final e que determina o comportamento do caso de uso é dado por

$final(p) := op_k$.

Sendo o resultado do caso de uso determinado pela última operação que é invocada, então a pós-condição do caso de uso é determinada pela pós-condição da última operação. Temos assim que $Post_{final(p)} \longrightarrow Post_{UC}$.

A pós-condição do caso de uso é o resultado de todas as pós-condições até à última operação mais a pós-condição desta. Se todas as pós-condições de um caminho no diagrama de estados, até à operação final, for designada por $Cond(p)$ então podemos dizer que:

$$Cond(p) \longrightarrow (Post_{final(p)} \longrightarrow Post_{UC})$$

ou de forma equivalente,

$$(Cond(p) \wedge Post_{final(p)}) \longrightarrow Post_{UC}$$

Se esta definição é verdade para um caminho no diagrama de estados, então podemos generalizar para todos os caminhos que se podem obter. Temos assim que:

$$\bigwedge_{p \in Sp}((Cond(p) \wedge Post_{final(p)}) \longrightarrow Post_{UC})$$

que corresponde a uma transformação de informação definida formalmente para uma notação informal (a pós-condição pode ser definida informalmente), registada em $Post_{UC}$.

É possível também descrever o que acontece quando se passa de uma descrição informal para uma formal. Nessas situações é necessário tornar explícita a informação que os utilizadores possuem, mas não a registam porque é implícita. Esta situação corresponde a dizer que $Post_{UC} \longrightarrow Post_{final(p)}$.

Se mais uma vez considerarmos a situação para todos os caminhos possíveis de obter no diagrama de estados temos que:

$$\bigwedge_{p \in Sp}((Cond(p) \wedge Post_{UC} \wedge Extra_{info(p)}) \longrightarrow Post_{final(p)})$$

, onde $Extra_{info(p)}$ corresponde à informação que é preciso tornar explícita e que usualmente é conhecimento do domínio da aplicação e dos utilizadores. Este é um passo importante do processo proposto, na medida em que constitui a justificação da mais-valia da introdução de rigor, ao induzir o levantamento de toda a informação anexa aos requisitos que deve ser tomada em linha conta no processo de desenvolvimento do sistema. A informação registada desta forma é útil para a fase de conceptualização, quer na definição arquitectural do sistema (de forma limitada) e na definição dos algoritmos das operações.

A utilização de diagramas de estado para descrever a execução de um caso de uso, faz com que a especificação do comportamento seja escrita de uma forma que torne claro quais são os fios de execução possíveis. O formalismo associado às máquinas de estado incute na descrição de comportamento características importantes de rigor que são desejáveis. Note-se que se poderia ter optado pela descrição do caso de uso recorrendo aos diagramas de actividade, mas estes não constituiriam uma base formal tão sólida como acontece com os diagramas de estado.

Por outro lado, a introdução da OCL no processo obriga a que se tenha que pensar de forma rigorosa, para que se torne possível descrever as condições associadas às pré, pós-condições e invariantes. A obrigatoriedade de ser concreto e rigoroso nas definições,

porque a linguagem assim obriga, faz com que informação implícita ou incompleta tenha de ser tornada concreta e precisa, para que o processo avance.

O processo que se apresenta, recorre ao uso de dois artefactos rigorosos, os diagramas de estado e a OCL, para disciplinar a passagem de conhecimento existente informalmente, ou parcialmente formalizado, para uma descrição de comportamento que tem todas as características necessárias a que possa ser encarada como uma peça de modelação confiável.

4.5.1 Máquinas de estados finitos

Da identificação dos traços que um diagrama de estado pode apresentar para a prossecução de um caso de uso, é possível recolher um conjunto de *strings* que correspondem à sequência de passos que se executaram no diagrama desde o estado inicial até um estado final. A recolha desta informação é importante para que se determine o que pode efectivamente acontecer na execução de um caso de uso, após se terem especificado as circunstâncias (estados e operações envolvidas) em que tal ocorre. É deste modo possível raciocinar ao nível dos traços de execução, bem assim como se consegue rigorosamente determinar quais são as condicionantes mais importantes para a execução do caso de uso. Note-se que essas condições já estão em grande parte identificadas nas pré-condições, mas a verificação dos traços de execução que é possível determinar, pode realimentar o processo de especificação dos requisitos e induzir o utilizador a ser mais preciso e em função dos traços e do conhecimento que tem do domínio do problema, acrescentar (ou alterar) restrições à execução.

Em casos de uso com muitos fios de execução, o tratamento formal dos traços, apresenta-se porventura como mais lucrativo que a pura inspecção por operacionalização.

A passagem da descrição dos casos de uso para o modelo dos diagramas de estado, como o processo propõe e está representado nas Figuras 4.4 e 4.11, faz com que a descrição do caso de uso proposta sob a forma de passos descritos informalmente, seja convertida para uma máquina de estados.

De forma a contextualizar a forma como constrói o diagrama de estados e como se obtêm os traços a partir dele, apresentam-se algumas definições que definem o processo pelo qual se conseguem determinar os fios de execução. Como um diagrama de estados é uma máquina de estados finitos, recorre-se à especificação destes artefactos de modo a definir o processo. Para utilizarmos os casos de uso e ser possível descrevê-los com outras técnicas é necessário que se demonstre que existe uma semântica comum na

qual se possam efectuar verificações e provas. Como nesta proposta se transformam os casos de uso em diagramas de estado, decorados com a informação dos contratos, é necessário fornecer uma definição formal de consistência dos dois modelos [Sinnig 07].

Essa definição formal de consistência serve também para demonstrar qual é o conjunto de implementações possíveis para uma determinada especificação. Ao definir-se formalmente a consistência do modelo proposto, consegue-se também determinar em que situações duas, ou mais, abordagens representam a mesma informação. Note-se que, por exemplo, a descrição das tarefas em CTT (ConcurTask Trees) [Paternò 00] utilizada normalmente para a descrição do diálogo interactivo pode ser formalmente consistente com o modelo dos casos de uso se ambos os mapeamentos para um outro formalismo forem consistentes e comparáveis nessa situação.

Seja a definição de uma máquina de estados finitos, dada pelas regras:

Definição 1: uma máquina de estados finitos é definida pelo tuplo $M = (Q, \Sigma, \delta, q_0, F)$, onde temos que,

- Q, é o conjunto finito de estados
- Σ, é o conjunto finito de símbolos que representam os eventos
- q_0, é o estado inicial, tal que $q_0 \in Q$
- F, é o conjunto final de estados, tal que $F \subseteq Q$
- $\delta : Q \times (\Sigma \cup \lambda) \rightarrow 2^Q$, é a função de transição entre estados, que determina o conjunto possível de estados dado um símbolo. λ representa a string vazia.

Para que seja possível comparar duas máquinas de estados é necessário definir mais algumas funções.

Definição 2: A operação de transição $\delta^* : Q \times \Sigma^* \rightarrow 2^Q$ é definida como sendo $\delta^*(q_i, w) = Q_j$, onde Q_j é o conjunto dos estados em que a máquina de estados finitos deterministicamente pode estar, tendo começado no estado q_i após a sequência de eventos w.

Definição 3: A função de aceitação $accept : Q \rightarrow 2^Q$, define o conjunto dos símbolos do alfabeto que podem ser aceites num determinado estado. É definida como $accept(q) = \{a | \delta^*(q, a)\}$.

Definição 4: A função $failure : Q \rightarrow 2^{\Sigma}$, devolve o conjunto dos símbolos que não podem ser aceites num determinado estado. A função $failure$ é o complemento de $accept$ e é definida como $failure(p) = \Sigma \setminus accept(p)$.

Definição 5: A linguagem aceite por uma máquina de estados é o conjunto de todas as strings de símbolos para os quais a função de transição permite chegar a um estado final. Cada elemento do conjunto representa um possível cenário da máquina de estados. A função é definida como $L(M) = \{w|\delta^*(q_0, w) \cap F \neq \emptyset\}$.

Definição 6: O conjunto de todos os traços gerados é o conjunto de todas as strings de eventos aceites no estado inicial pela função de transição. A sua definição é $traces(M) = \{w|\delta^*(q_0, w)\}$.

As definições apresentadas garantem que de um ponto de vista formal é possível, dada uma máquina de estados, prever o seu comportamento e determinar o conjunto de passos que uma determinada entidade, sujeita a estímulos exteriores, executa durante o seu ciclo de vida (ou uma parcela do mesmo). Logo, é possível que depois de ter o caso de uso descrito desta forma, a equipa de projecto possa saber com exactidão qual é a sua expressão de comportamento. Apenas existirá falta de formalismo e ambiguidade se não tiver sido possível a criação de um diagrama de estado a partir da descrição do caso de uso. Se tal acontecer, deve-se ao facto de a equipa de projecto desconhecer qual é o comportamento do caso de uso e não conseguir materializá-lo numa descrição de comportamento não ambígua.

Transformação dos casos de uso para máquinas de estado

Importa, nesta altura, formalizar o processo que permite que a interacção definida por um caso de uso possa ser representada como uma máquina de estados, de modo a que possa posteriormente ser animada, ou sujeita a raciocínio formal, consoante a intenção da equipa de projecto.

Um caso de uso é constituído pelos vários passos da sua descrição, sendo estes tanto do cenário principal como dos cenários alternativos. Os vários passos algorítmicos de um caso de uso podem representar uma sequência de acções ou uma escolha.

Em abstracto, cada passo do caso de uso pode ser representado como uma máquina de estados finita. A máquina de estados é composta por um estado inicial - o princípio do caso de uso - e vários estados finais a que se chega através de eventos, como está

ilustrado na Figura 4.16. Cada evento representa um resultado observável diferente.

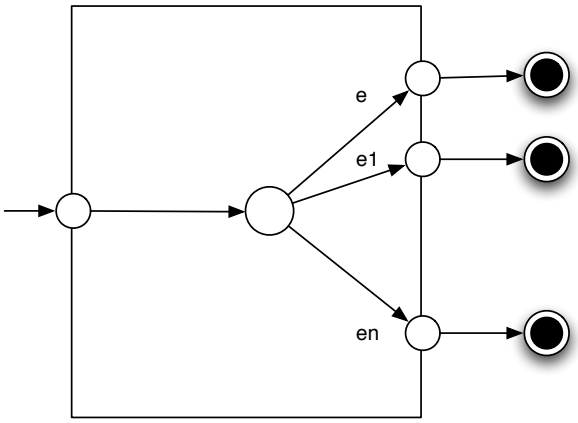


Figura 4.16: Uma máquina de estados para um caso de uso.

Sendo p_1 e p_2 passos do caso de uso, com esta abordagem é possível criar uma sequência de máquinas de estado para representar o caso de uso, sendo apenas necessário que se crie uma transição entre os estados finais da máquina de estados M_1 do passo algorítmico p_1 e a máquina de estados M_2 do passo algorítmico p_2, conforme se apresenta na Figura 4.17.

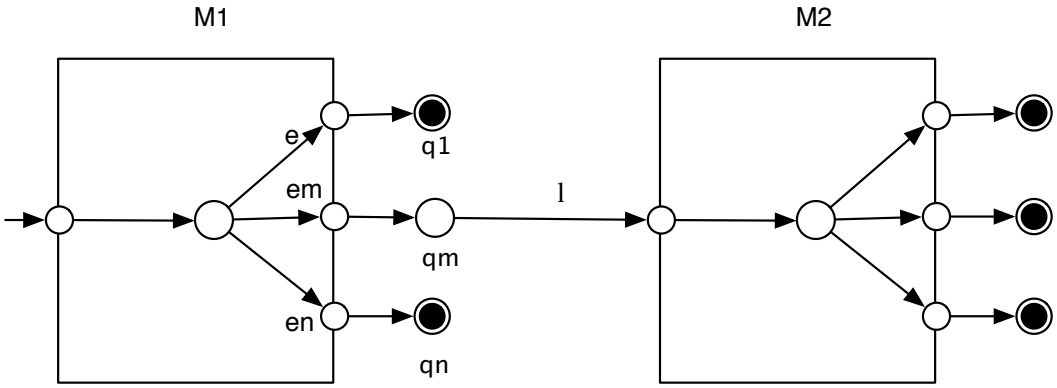


Figura 4.17: Sequência de máquinas de estado.

Exemplificando para o caso de uso da Figura 4.15, que corresponde ao caso de uso "Levantar Dinheiro" numa máquina ATM. Considere-se dessa descrição o terceiro passo, que implica que "O sistema verifica se o cliente tem dinheiro suficiente na sua conta e se o limite diário de levantamento não foi excedido", segundo o que foi descrito. Na passagem da descrição deste passo do caso de uso para a máquina de estados, considere-se que daqui resultam três transições possíveis: uma quando não existe saldo

na conta, outra quando o montante pedido faz com que se exceda o montante disponível diariamente e uma terceira transição que é de sucesso e que conduz ao próximo passo do cenário normal.

Podemos então mapear este passo da descrição na máquina de estados que se apresenta na Figura 4.18.

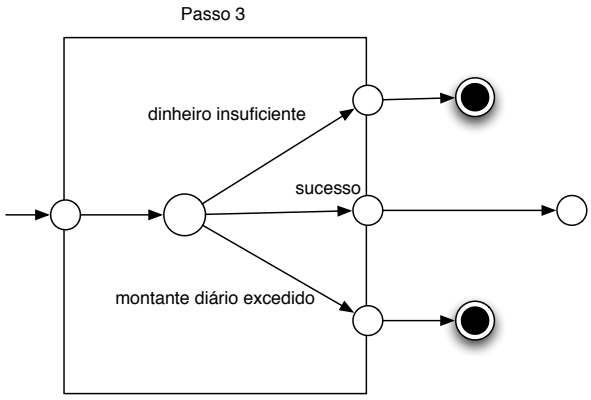


Figura 4.18: Máquina de estado para o passo 3 da descrição do caso de uso.

O caso de sucesso, faz a ligação à máquina de estado do próximo passo da descrição que corresponde à autorização de levantamento, conforme foi expresso na narrativa do caso de uso.

Esta transformação é exaustiva e no fim do processo o caso de uso é descrito por uma sequência de máquinas de estado, que podem ser simplificadas de modo a produzirem um diagrama mais fácil de ler. Com a aplicação exaustiva destas transformações uma narrativa bem descrita dará origem a um diagrama de estados, sendo que o analista com o conhecimento que tem do sistema e do formalismo das máquinas de estado pode permitir-se colapsar alguns dos estados intermédios.

É pois possível transformar os casos de uso em diagramas de estado, sendo que se possibilita também que se de outras representações for possível o mapeamento para máquinas de estado finitas, é possível demonstrar se dois casos de uso descritos de forma distinta correspondem a conceitos similares. Nesse sentido, por exemplo, é possível explorar a capacidade de representação dos diagramas de tarefas representados em CTT [Paternò 00] e raciocinar sobre os casos de uso que os modelam no âmbito do Unified Process. A expressividade dos modelos expressos em CTT, que permitem modelar o diálogo interactivo, pode ser comparada com os modelos de casos de uso, se a modelação de tarefas for expressa em termos de máquinas de estados finitos.

No entanto, para que seja possível comparar duas máquinas de estado geradas a

partir de diferentes representações, e de forma a estabelecer que é possível utilizar, com igual capacidade representativa, os diagramas de caso de uso é necessário que se estabeleçam algumas noções de consistência para que o raciocínio seja válido.

Essas noções de consistência, devem levar a que se estabeleça uma definição para duas transformações para máquinas de estados finitos.

Definição Consistência: se $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ for a máquina de estados que representa um caso de uso U e $M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ a máquina de estados que representa um modelo numa outra notação N, pode afirmar-se que a representação N é consistente com o caso de uso U se for verdade que:

1. existe inclusão da linguagem de N em U, tal que $L(M_2) \subseteq L(M_1)$
2. a cobertura dos traços das duas máquinas de estado é suficiente, sendo que $\forall n \in N, N = \{traces(M_2) \setminus L(M_2)\}$
   (a) seja $Q_{M1} = \{p_1, p_2, \ldots, p_n\}$ seja $\delta^*(q_{01}, n)$. Todos os p_i, são apenas os estados que podem ser alcançados a partir do estado inicial de M_1, depois de ter aceite n.
   (b) seja $Q_{M2} = \{q_1, q_2, \ldots, q_m\}$ seja $\delta^*(q_{01}, n)$. Todos os p_i, são apenas os estados que podem ser alcançados a partir do estado inicial de M_2, depois de ter aceite n.
   (c) é necessário que $\forall p_i \in Q_{M1} \exists q_j \in Q_{M2}. failure(p_j) \subseteq failure(q_i)$

Como pré-condição desta definição, é necessário que as máquinas de estado operem sobre o mesmo alfabeto.

Com o mapeamento atrás apresentado, conseguiu-se demonstrar que a utilização de diagramas de estado para a representação dos casos de uso é válida, bem assim como permite que através da transformação de modelos, se possam reduzir (transformar) outros modelos em diagramas de estado.

Apesar de poder ser sistematizada, esta transformação de casos de uso para os diagramas de estado não é trivial, e implica um conhecimento aprofundado do domínio do problema. Este salto semântico é um dos grandes desafios dos modelos de processo, visto que a construção do sistema é feita a partir de descrições que detalham a interacção entre o sistema e o actor que invoca o caso de uso. A descoberta das actividades que alteram o estado da aplicação, bem como as pré e pós-condições e o invariante do sistema são dados importantes para a construção da arquitectura interna do sistema. Esta é também uma das justificações para a utilização da OCL nesta fase

do processo. Como a linguagem está assente em premissas rigorosas, só é possível nela descrever informação que esteja devidamente assente e estruturada. Por outro lado, mesmo que essa informação ainda não esteja completamente definida, a obrigação de criar a especificação rigorosa em OCL obriga a equipa de projecto a procurar respostas e a extrair informação do cliente.

A Figura 4.19 demonstra graficamente o processo de construção da expressão de comportamento de um caso de uso a partir da sua narrativa textual.

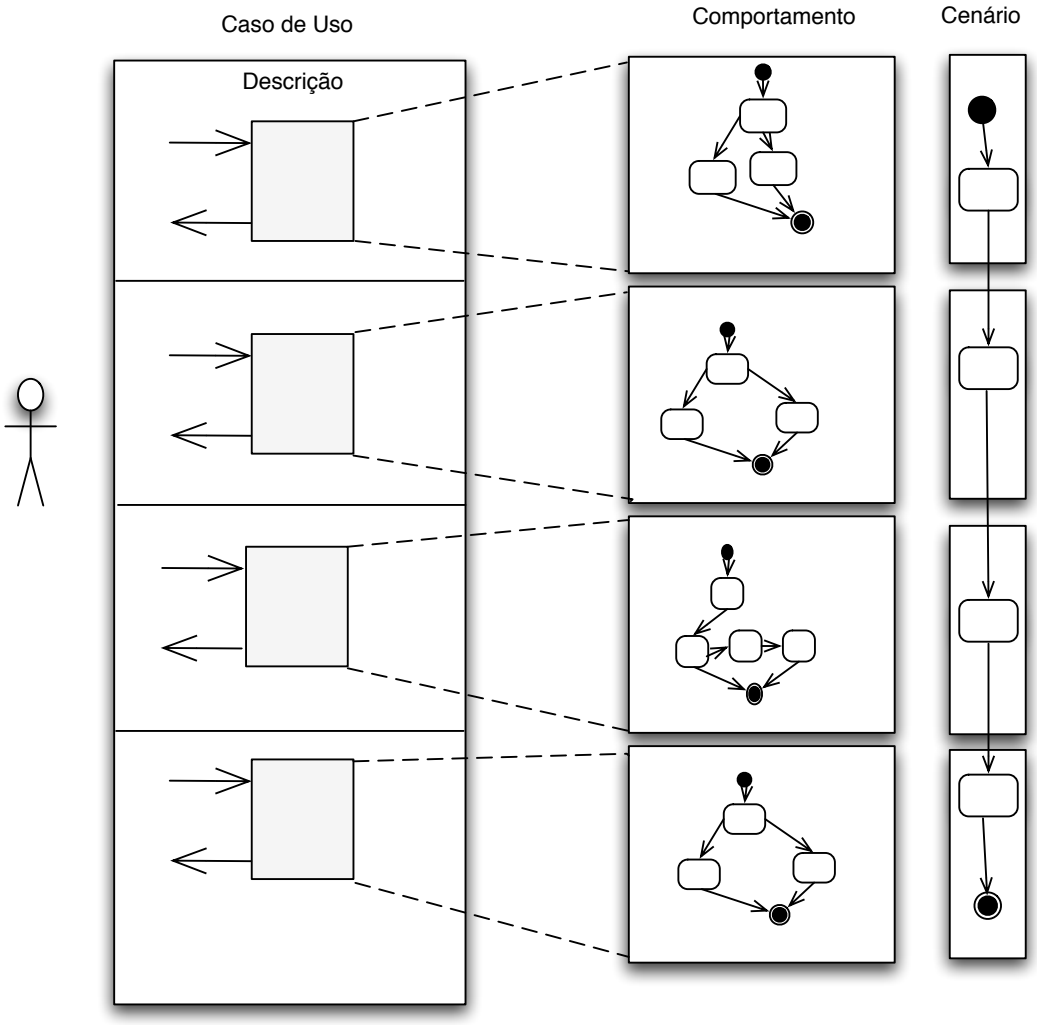


Figura 4.19: Transformação de um caso de uso na sua expressão de comportamento.

Uma vez que consideramos que um caso de uso é uma sequência de passos, a narrativa do caso de uso definida de forma textual estabelece a ordem pela qual esses passos são invocados. Estes passos seguem o padrão genérico apresentado na Figura 4.7

e correspondem a momentos de interacção entre o actor e o sistema. A descrição textual do caso de uso já organiza estes momentos de interacção e coloca-os pela ordem que o cliente definiu. A construção do diagrama de estados que especifica o comportamento do caso de uso é obtido através da especificação sucessiva do comportamento de cada um dos passos transaccionais que o constituem.

Cada um dos passos da narrativa pode ser decomposto numa expressão de comportamento que corresponderá a uma pequena máquina de estados. Esta máquina pode ser mais ou menos complexa consoante o detalhe que o cliente tenha colocado na descrição textual. O processo descreve para todos os passos do caso de uso a sua expressão de comportamento com recurso a uma máquina de estados que contempla os casos de sucesso e os casos de insucesso. Na descrição textual, feita de acordo com o arquétipo textual, apresenta-se primeiro a sequência de passos do cenário normal e depois identificam-se os cenários alternativos tendo o cuidado de referenciar o número de ordem do passo a que se referem. Quando se descreve a expressão de comportamento de um passo do caso de uso devem especificar-se todas as alternativas possíveis, sendo estas pertencentes ao cenário positivo ou aos alternativos.

Após todos os passos do caso de uso terem sido transformados nos correspondentes diagramas de estado, obtém-se o diagrama de estados que modela o comportamento do caso de uso. É possível nesta fase escolher o nível de detalhe que se pretende mostrar, visto que o diagrama resultado pode ser bastante simples se recorrer ao mecanismo de decomposição hierárquica e representar cada uma das máquinas de estados dos diferentes passos do caso de uso como sendo um estado do diagrama mais geral.

Tomando em consideração o diagrama de estados que é obtido pela composição dos vários contributos, é possível raciocinar sobre os cenários existentes no caso de uso, e determinar quais são as interacções possíveis. Tal como se representa na Figura 4.19, um cenário corresponde a uma travessia no diagrama de estados, em que apenas alguns dos estados são visitados. A escolha dos estados que são visitados na execução de um cenário deriva das expressões lógicas que determinam qual é o comportamento em cada uma das máquinas de estados de cada um dos passos do caso de uso.

Com base neste pressuposto, pode ser alargada a definição de contrato que tinha sido definido para um caso de uso e aplicada aos vários passos desse caso de uso. Considera-se que um passo de um caso de uso possui também uma pré e uma pós-condição e que o comportamento da sua máquina de estados é reflexo de tais asserções. Cada uma das sub-máquinas de estado terá assim uma expressão de comportamento igualmente regulada por expressões escritas em OCL.

Os cenários são descritos posteriormente com recurso aos diagramas de sequência

de forma a registarem a interacção entre as entidades que tornam possível a execução do caso de uso. Os diagramas de sequência utilizam entidades que são do modelo de domínio e outras que são já elementos de concepção, mas que são necessárias à descrição da interacção. As expressões OCL são colocadas também como anotações dos diagrama de sequência, e além de regularem o fluxo de execução podem já referir restrições que são derivadas da arquitectura da solução que será a fase seguinte do projecto.

A informação relativa à especificação OCL que é transposta para o modelo, e necessariamente mais tarde alimenta o diagrama de classes, é ela também obtida de forma iterativa através da execução do processo de construção do diagrama de estados para cada caso de uso, como se referiu anteriormente.

Dessa forma pode afirmar-se quando se descreve a interacção que:

- o contexto de cada restrição expresso em OCL é o objecto instância da classe que assegura o controlo da execução do caso de uso;
- uma pré e pós-condição é definida para cada tipo de informação trocada pela entidade que controla a execução e as outras entidades. Consoante esta troca constitua um ganho ou uma perda para o objecto que controla a execução, é necessário que sejam especificadas as pré e pós-condições que regem o relacionamento entre as diversas entidades.

  No exemplo do levantamento de dinheiro na máquina ATM, é necessário que se especifiquem as restrições que condicionam a execução do caso de uso, sendo que associadas às transições existem condições que é necessário testar. No levantamento de dinheiro é necessário que o diagrama de estados contemple, na sua construção, que expressões como:

```
pre: self.saldo > 0
pre: valor < montanteDiarioPermitido

post: self.saldo = self.saldo@pre - valor
post: historicoLevantamentos =
         historicoLevantamentos@pre -> including(valor)
```

  representam pré e pós-condições que a lógica de negócio impõe. Estas expressões devem ser materializadas no diagrama de estados, seja nas transições seja nos próprios estados.

- para cada restrição encontrada, o algoritmo deve promover que também as relações semânticas existentes com outros objectos do domínio sejam especificadas. Estas

relações não derivam do diálogo que directamente se estabelece com o objecto de controlo, mas são resultado da informação de negócio fornecida ao nível do modelo de domínio. Estas restrições são na maior parte dos casos anotações às associações que existem entre entidades do modelo de domínio e que serão transpostas para o diagrama de classes. Um exemplo deste tipo de restrições seria a especificação dos requisitos dizer que um determinado cliente do banco apenas poderia efectuar um número N de levantamentos. Essa restrição seria escrita em OCL e constituiria uma pré-condição na execução do caso de uso, mas é necessário dizer no modelo de domínio que a multiplicidade existente será do tipo `0..N`, e

- as expressões que se materializam como invariantes de estado são também actualizadas com a informação das restrições anteriormente descobertas.

Como referido em [Roussev 03], este é um processo sujeito a realimentação (por *backtracking*), porque começando o sistema a ser detalhado logo na sua componente comportamental, a informação relevante é descrita nas fases iniciais do processo, permitindo assim que em comparação com outras abordagens de decomposição das tarefas de análise de um sistema, se possa considerar que este método assegura uma grau de estabilidade que está menos sujeito ao natural processo iterativo e com regressões, que tipicamente caracteriza estas actividades.

A abordagem proposta pretende assegurar que entre a especificação de um caso de uso e a correspondente concretização, primeiro na vertente de análise e depois na consequente construção, não existem os tradicionais problemas de consistência entre estes dois planos. Reforça-se assim o que, para Jacobson [Jacobson 92], é uma medida do sucesso na construção de sistemas, nomeadamente a formalização das especificações dos casos de uso e o desenvolvimento da solução através da transformação de modelos. Além do mais esta estratégia permite que o discurso seja mantido a um nível suficientemente elevado de abstracção e centrado nos objectos de negócio que são envolvidos no diálogo especificado pelo caso de uso. Este facto é uma vantagem óbvia que permite que a especificação do caso de uso seja feita com os decisores do negócio (o cliente e os utilizadores principais) e que seja claramente perceptível onde acaba a análise e onde começa a elaboração da solução.

4.6 A descrição do método

Importa antes de apresentar um caso prático de utilização do processo proposto explicitar as premissas em que se baseia o método de aplicação do processo e quais são os resultados esperados de cada uma das suas fases. A Figura 4.20 apresenta de forma sumária as principais etapas do processo que é proposto.

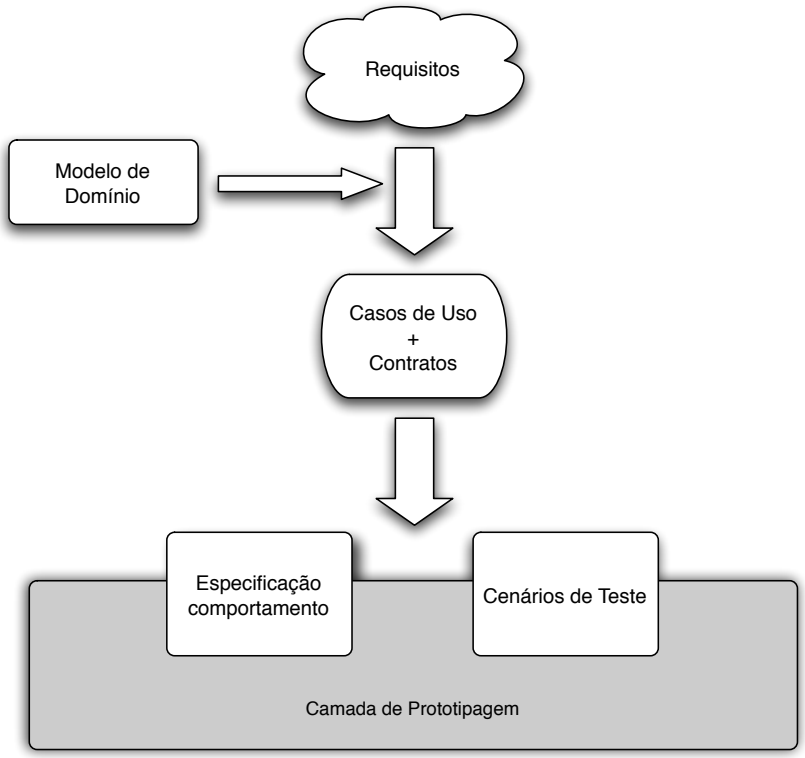


Figura 4.20: Metodologia de aplicação do modelo de processo.

Num primeiro momento os requisitos são comunicados à equipa de projecto que conjuntamente com o cliente elabora o modelo de domínio, no qual se descrevem quais são as principais entidades de negócio existente, se define o vocabulário e se explicitam os relacionamentos semânticos entre as entidades. Com esta informação a equipa de projecto elabora os casos de uso, criando o respectivo modelo e as descrições textuais dos mesmos, e especifica a informação relativa às pré e pós-condições e invariante.

Num segundo momento, a especificação do comportamento do caso de uso é descrita com o recurso à elaboração da máquina de estados que rege o seu desempenho e onde se identificam os eventos e condições que regulam o funcionamento da mesma. Nesta fase é possível estabelecer os objectivos de teste que se pretendem descrever, permitindo-se

associar mais do que um caso de uso a um teste, bastando para tal compor as máquinas de estado necessárias. A informação dos objectivos de teste, pode ser refinada para a elaboração dos cenários de teste que permitem concretizar e validar as interacções iniciadas pelo actor e que despoletam a resposta do sistema. Os artefactos de teste desta segunda fase do modelo podem ser animados com o recurso a uma camada de prototipagem, baseado em objectos activos, e que possibilita a operacionalização dos testes e a consequente validação dos requisitos, ainda na fase de análise.

Como se referiu, o processo apresenta como principal mais-valia o facto de permitir elaborar, ainda na fase de análise de requisitos, uma série de testes aos elementos de análise o que possibilita obter mais facilmente a validação do cliente, bem como efectuar a recolha de informação relevante para as fases subsequentes do projecto. É claro que, a efectiva qualidade dos testes que se podem promover é razão directa da qualidade com que são definidos as expressões rigorosas que anotam os casos de uso e que determinam os cenários que se pretendem explorar. Quão melhor fundada e solidamente expressa for essa informação, mais úteis serão os testes aos requisitos que se podem conduzir nesta fase do projecto.

O processo apresentado permite que se possam identificar como distintivos os seguintes itens:

- os requisitos funcionais são expressos sobre a forma de casos de uso enriquecidos com a noção de contratos, isto é, com informação sobre as pré e pós-condições e respectivo invariante;
- a identificação dos vários cenários de execução de um caso de uso é capturada, possibilitando-se quer o raciocínio formal sobre os possíveis fios de execução, quer a correspondente animação por prototipagem para validação operacional, e
- a partir da informação dos cenários é possível a identificação das trocas de mensagens entre os diversos actores e o sistema, permitindo a criação de casos de teste que são especificados através de interacções, ou seja, com o recurso a diagramas de sequência.

Com a informação existente na fase de análise e com a presença dos principais interessados no resultado do projecto, é possível lidar com um problema de escala no que diz respeito ao conjunto de fios de execução possíveis de obter a partir da explicitação dos vários cenários dos vários casos de uso. Os utilizadores são essenciais nesta fase, de modo a identificarem quais são as diversas alternativas de fios de execução que merecem ser testadas e quais são aquelas que não acrescentam mais-valia ao teste

de requisitos que se pretende promover. Caso o raciocínio sobre os fios de execução seja conduzido por uma ferramenta de *model-checking*, esta poderá ter de lidar com problemas de explosão combinatorial dos estados internos do modelo de simulação, pelo simples facto de não possuir informação sobre quais são os caminhos relevantes.

A construção de um modelo de casos de uso com a informação dos contratos não é uma tarefa trivial. A definição declarativa das expressões relativas aos contratos obriga a que o analista seja preciso e muito rigoroso. A semântica de cada uma das expressões relativas aos contratos deve ser bem definida e consistente. O conjunto das expressões rigorosas é um complemento da documentação de projecto iniciada com o modelo de domínio e, tal como este, define de forma muito evidente o vocabulário e as condicionantes dos requisitos[7].

Uma vez descritos os contratos e identificadas as restrições que a execução dos vários cenários dos casos de uso implicam, é possível simular a execução dos mesmos. Esta simulação, como se referiu, pode ser feita com recurso a ferramentas de verificação de modelos, ou então, através de mecanismos de prototipagem. Nesta fase é possível determinar quais são os parâmetros que definem o estado interno do sistema aquando da execução de um determinado caso de uso, isto é, é possível instanciar os casos de uso que se pretendem animar de modo a verificar, tendo em linha de conta os parâmetros fornecidos, qual é o fio de execução que é seguido. Esta fase permite ao analista, bem assim como ao cliente e utilizadores, efectuarem a validação dos requisitos e perceber se o que havia sido previamente especificado corresponde ao que se pretende ver desenvolvido e em funcionamento. Através da verificação, quer esta seja formal ou operacional, compara-se se o comportamento obtido é, ou não, diferente do comportamento esperado.

Os objectivos de teste possibilitam que se afira o comportamento que o sistema software exibirá quando determinado caso de uso for invocado, permitindo aos elementos envolvidos na captura de requisitos a validação desse mesmo comportamento. No entanto, para que mais informação sobre o sistema possa ser obtida é necessário aproximar os cenários de teste do modelo de programação subjacente. Assim é necessário transformar os objectivos de teste em casos de teste para cada um dos cenários possíveis de serem considerados. De modo a permitir descrever as interacções que existem associadas aos casos de teste, utilizam-se os diagramas de sequência para detalhar as entidades envolvidas e as respectivas trocas de mensagens entre elas. Cada um destes diagramas descreve como é que o actor (os futuros utilizadores) interage com o sistema

[7] É determinante que a utilização dos mesmos termos seja comum quer ao modelo de domínio quer às expressões formais relativas aos contratos.

e qual é a resposta deste e quais são as mensagens que suportam esta interacção.

A utilização de diagramas de sequência possibilita que se faça a ponte com os tipos de dados que estão envolvidos na construção do sistema e que correspondem à transformação das entidades do modelo de domínio para as classes, os tipos de dados, do diagrama de classes. Os cenários de teste, vistos sobre a forma de diagramas de sequência e sendo o resultado de instanciar os parâmetros formais dos casos de uso com os valores pretendidos permitem efectuar diferentes tipos de testes. Entre esses testes podem-se identificar os que são de natureza funcional, os testes funcionais, e os testes de robustez. Os casos de testes funcionais correspondem às situações em que se instancia o cenário de teste com o cenário normal dos casos de uso envolvidos, de forma a verificar se o comportamento obtido é o esperado, enquanto nos testes de robustez o que se pretende testar é se as restrições que foram associadas a cada caso de uso são eficazes. Estes testes destinam-se a verificar a correcção dos cenários alternativos dos casos de uso.

A Figura 4.21 sintetiza os contributos que o modelo de processo proposto fornece às fases seguintes do projecto de construção do sistema software.

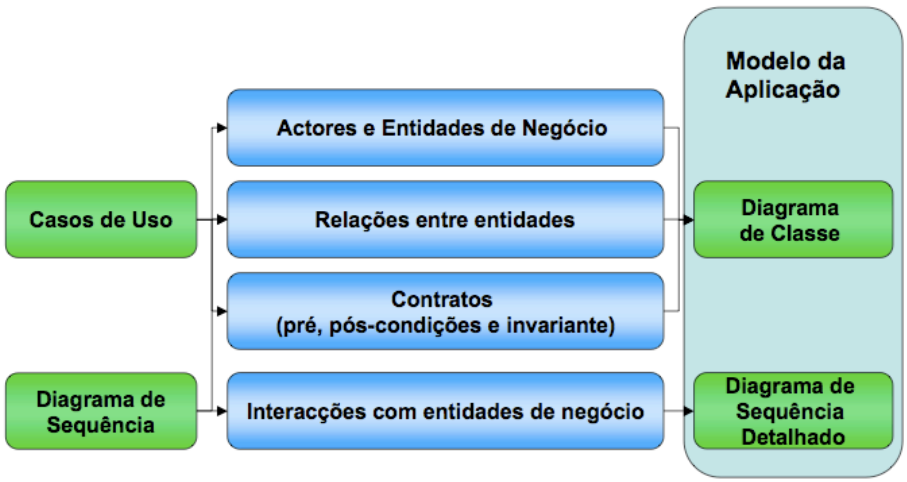


Figura 4.21: Contributos do processo para as fases seguintes o projecto.

A informação que se obtém por refinamento sucessivo e contínuo esforço de modelação possibilita que a fase de análise seja muito relevante para as fases de concepção e desenvolvimento. Os resultados obtidos, se seguido o processo, pela fase de análise são elementos de modelação das fases seguintes. A especificação dos casos de uso, com a consequente identificação das entidades de negócio envolvidas, a descrição das relações entre essas entidades e a informação relativa aos contratos que devem ser garantidos a

nível da camada de negócio, alimenta directamente o diagrama de classes do modelo da aplicação.

A especificação comportamental dos casos de uso, permite a descoberta de operações e restrições funcionais e estruturais que o diagrama de classes do modelo da aplicação tem de incorporar.

Idêntica situação se passa com os diagramas de sequência que são utilizados para descrever as interacções que permitem descrever um fio de execução do caso de uso. A sua elaboração prevê a interacção entre entidades do modelo do domínio, mas pode tornar-se necessário que para permitir descrever o comportamento tenham de ser identificadas outras entidades que resultam da necessidade de especificação do fluxo de troca de mensagens. Um exemplo típico de entidades que são colocadas nos diagramas de sequência e não pertencem ao modelo de domínio são aquelas que desempenham o papel de interface com o utilizador. Estas entidades são já do modelo da aplicação, mas foram detectadas na fase de análise porque se tornaram necessárias para descrever a interacção que suporta os requisitos. Outras entidades, como colecções ou mapeamentos de entidades do modelo de domínio são tipicamente descobertas pela descrição sistémica dos diagramas comportamentais que modelam um caso de uso.

4.6.1 Operacionalização dos casos de uso

Os casos de uso correspondem às macro-funcionalidades que o sistema oferece [d'Souza 99], sendo que o correspondente diagrama de casos de uso representa uma vista global que descreve as funções primárias do mesmo. No entanto, os casos de uso não se reflectem como funções do sistema de forma linear e automática. É inclusive uma das grandes dificuldades dos analistas a necessidade de não associarem mentalmente e ao nível da concepção do modelo, os casos de uso a pontos de entrada básicos do sistema (opções de menu, botões, etc.).

Para efeitos de prototipagem e validação dos resultados é necessário que se determinem quais as acções que dão origem à execução de um caso de uso, de forma a desenvolver os cenários de teste. Como os casos de uso são anotados com a informação relativa a pré e pós-condições torna-se possível utilizar esta informação para obter a ordem parcial das funcionalidades que o sistema oferece na execução de uma tarefa. Consegue-se assim obter a sequência de casos de uso que permitem a criação dos cenários de teste e validação. A descrição destas sequências de casos de uso que constituem uma tarefa permitem efectuar uma avaliação dos requisitos, possibilitando que estes sejam revistos e alterados antes de nova ronda de validação e prototipagem.

Para que efectivamente seja possível efectuar prototipagem e validar as tarefas é necessário acrescentar mais informação do que aquela que se obtém através da ordenação parcial dos casos de uso. Por refinamento, através da utilização de outros diagramas UML, como sejam os diagramas de actividade, estado e sequência, consegue-se colectar informação adicional que versa as trocas de mensagens entre objectos. O diagrama de sequência final torna-se assim a narrativa do cenário de teste que permite ao utilizador validar as tarefas que o sistema suporta. Ao antecipar grande parte da modelação comportamental, de forma a testar os requisitos, o analista efectua uma parcela substancial do trabalho de modelação, com uma mais-valia evidente.

Nesta fase, trabalham-se as expressões lógicas que representam as asserções e raciocina-se sobre o estado interno do sistema e sobre a sua coerência. Esta proposta é passível de ser integrada num modelo de processo de Engenharia de Software baseado em UML. Os requisitos funcionais são descritos através da elaboração dos casos de uso e a informação dos contratos é elaborada numa linguagem rigorosa, com as pré e pós-condições e os invariantes a serem descritas em OCL.

Numa iteração seguinte do processo, o analista em conjunto com os utilizadores, descreve as tarefas importantes do sistema e identifica os cenários relevantes do ponto de vista externo. Com este detalhe ao nível da modelação dos requisitos é possível prototipar os cenários e validar os requisitos. O processo proposto apresenta como principais características:

- definição dos casos de uso - permite associar aos casos de uso a informação das asserções que podem ser validadas por prototipagem e verificadas por ferramentas próprias;
- enriquecimento do resultado da fase de análise - porque se combinam vários diagramas de forma a criar uma definição do cenário de teste dos requisitos. A criação do diagrama de estado é essencial para a especificação correcta do caso de uso e permite o envolvimento dos utilizadores na altura certa do processo de análise. Em sistemas interactivos a especificação da camada interactiva pode ser feita juntamente com a captura de requisitos, para promover a identificação dos eventos e métodos da interface com o utilizador, como descrito em [Ribeiro 07, Campos 06], e
- descoberta de aspectos relativos ao funcionamento - porque existem condicionantes que estão directamente ligados ao ambiente em que se prevê (ou se sabe) que o sistema vá funcionar. A especificação de como é feita a comunicação entre as entidades (sequência, concorrência ou paralelismo), de que tipo é o canal de

comunicação (síncrono ou não) é importante para a validação dos requisitos. Ao se descrever a sequência de acções que ilustram a execução da tarefa, o analista pode etiquetar e modelar estas características do diálogo entre as entidades. Se a plataforma de prototipagem possuir mecanismos que permitam a concretização destas interacções, a validação é desta forma mais efectiva.

Operacionalização da validação de requisitos

Como se referiu anteriormente, os cenários de validação dos requisitos são obtidos a partir dos casos de uso e devem ser descritos como uma sequência válida de invocações e de resultados. Em função dos resultados apresentados é possível determinar se o teste foi ou não satisfatório. No que concerne aos resultados que são possíveis de obter, estes podem ser de três tipos:

- resultado negativo - que surge quando uma pós-condição não é respeitada durante a execução do cenário, o que significa que o sistema fica num estado incoerente;
- resultado positivo - quando o cenário de teste é concluído na totalidade o que implica que todos os testes que foram efectuados foram verdadeiros, e
- resultado inconclusivo - quando a falha na avaliação de uma pré-condição impede que o resto do cenário seja testado. Não é líquido que nestas situações se possa considerar que o resultado do teste deva ser negativo, visto que tal pode ter acontecido propositadamente (por se querer verificar que a pré-condição é respeitada), ou por a sequência de casos de uso que pode estar a ser testada não corresponder a uma tarefa válida no sistema, ou então, porque os contratos especificados para os casos de uso envolvidos não estarem correctamente especificados.

Os diagramas de sequência são obtidos a partir do diagrama de casos de uso e da sua descrição, decorado com a informação das asserções (pré e pós-condições) e com a informação adicional descrita com outros diagramas comportamentais (actividade e/ou estado) que explicitam a forma como os actores interagem com o sistema e produzem efeito no seu estado interno. O diagrama de sequência que é utilizado para a validação do cenário de teste não corresponde ao diagrama de sequência que é possível obter numa fase de modelação mais avançada, mas constitui uma aproximação muito significativa a este, à excepção de algumas definições que são derivadas do processo de construção e que são dependentes do modelo de programação utilizado. Como nesta fase não se obteve um diagrama de classes completamente definido (o que só acontecerá na fase

de concepção arquitectural do sistema), o diagrama de sequência pode conter alguns conceitos (tipicamente entidades) que não farão parte da modelação final. Esta etapa da modelação que se destina fundamentalmente à recolha de requisitos e posterior validação operacional, pode apresentar entidades que são em grande parte actores ou entidades de negócio, retirados do domínio da aplicação, que podem não existir como tipos no diagrama de classes final. Uma vantagem que o diagrama de sequência que representa o caso de uso apresenta, corresponde ao facto de nesta altura ser possível detalhar a informação sobre os protocolos de comunicação utilizados, o que permite representar desde muito cedo na análise a forma como os dados são trocados e como é que as entidades se relacionam.

O diagrama de sequência tem desta forma mais informação que o correspondente caso de uso, na medida em que relaciona outros elementos do modelo e é mais rico e detalhado que o caso de uso que lhe deu origem (já com a informação relativa aos contratos - pré e pós-condições). A informação relativa às asserções que descrevem os contratos é descrita em OCL se descrever restrições que relacionem elementos do modelo, ou em expressões lógicas se estão relacionadas com restrições aos dados (parâmetros) ou à descrição do fluxo de controlo.

O diagrama de sequência que suporta o cenário de teste do caso de uso pode englobar tanto o cenário normal, como os cenários alternativos (nomeadamente os de excepção). Tendo em linha de conta a complexidade de desenho dos diagramas de sequência, incrementada na UML 2.0 com a introdução das frames de interacção, em vez de se ter um diagrama de sequência que descreva o cenário principal e os cenários alternativos, estes podem ser separados em diagramas distintos, também de forma a se reduzir a complexidade do cenário de teste.

4.7 Validação Operacional

As actividades de modelação que se empreendem no processo de especificação e detalhe do caso de uso, conduzem à informação que pode ser operacionalizada, de forma a ser constituída como um cenário de teste.

A validação dos sistemas software faz-se em grande medida pela prossecução dos testes feitos com o sistema já em fase estável para que os testes façam algum sentido. Tendo em conta a descrição das actividades do Unified Process, verifica-se (como seria expectável) que existem diversos pontos ao longo da fase de desenvolvimento aonde se efectuam tarefas de teste. Esses testes são numa primeira fase testes unitários e

posteriormente testes de integração. É possível que o cliente, ou os utilizadores, sejam envolvidos nestes testes, embora em grande parte das situações tal só aconteça numa fase mais avançada do projecto de desenvolvimento. Como em boa medida os utilizadores estão afastados das fases mais tecnológicas, é na fase de análise e levantamento dos requisitos que podem ter um papel mais activo.

Com a proposta de processo de modelação que este capítulo apresenta, o utilizador é envolvido na descrição da informação rigorosa que sustenta as pré e pós-condições e na especificação do fluxo da tarefa que se pretende detalhar. Neste contexto o utilizador está em condições para poder validar os requisitos de duas formas possíveis:

- através da definição do conjunto de traços e com base em raciocínio ao nível das strings geradas, verificar se o comportamento especificado é o pretendido, e
- através de inspecção operacional dos resultados apresentados pelo protótipo que é possível construir a partir do diagrama de estados e do diagrama de sequência.

No diagrama de estados estão presentes os diversos estados pelos quais passa o caso de uso. Cada um desses estados corresponde, do ponto de vista da operacionalização a uma vista sobre o estado interno da execução da tarefa. Da visualização do diagrama de estados é possível criar uma família de objectos interligados entre si e que permitam simular a passagem de controlo entre os diversos estados. O utilizador em função dos dados que introduz no protótipo e da visualização que faz do estado em que se o caso de uso se encontra, avalia a execução e valida se tal corresponde ao que efectivamente pretende. Como a operacionalização é feita recorrendo a uma plataforma de prototipagem que permite escolher o tipo de diálogo entre os objectos, o utilizador pode a este nível também determinar como é que as diversas transições de estado são efectuadas. É nesta fase que se explicitam os requisitos que derivam do facto de o sistema que se está a modelar ter características de tempo, de concorrência ou de distribuição. Por exemplo, esta fase de prototipagem é muito importante se utilizada para validar a camada interactiva, que frequentemente tem problemas que derivam da existência de restrições de concorrência e tempo, e permitir ao utilizador, e à equipa de projecto, a identificação dos eventos da interface e a forma como eles são tratados.

Se o diagrama de estados é determinante para se validar e inspeccionar o estado em que o caso de uso se encontra, o diagrama de sequência apresenta a troca de mensagens entre o sistema e os utilizadores, com o intuito de detalhar algoritmicamente a tarefa que se pretende efectuar. O diagrama de sequência é obtido a partir dos casos de uso e especifica a ordem pela qual as mensagens são enviadas e determina quem responde. A combinação da informação do diagrama de sequência com o detalhe do diagrama

de estados permite que se prototipe a troca de mensagens, bem como permite que se associe a pontos específicos da troca de mensagens a informação sobre o estado em que se encontra o caso de uso e quais são as condicionantes em cada um desses estados.

Ao nível do diagrama de sequência o utilizador pode especificar que algumas trocas de mensagens podem obedecer a um comportamento não síncrono, sendo necessário, que a arquitectura de prototipagem seja capaz de operacionalizar esses requisitos.

A plataforma de prototipagem que se detalha no capítulo seguinte, permite que os utilizadores possam, com a ajuda da equipa de projecto, animar a tarefa que pretendem especificar e para a qual descreveram os requisitos. A plataforma baseia-se numa colecção de objectos activos que trocam mensagens entre si e que podem estar organizados segundo diversos tipos de ligação. Cada ligação possibilita um diálogo protocolar próprio entre um objecto e todos a que está ligado. Possibilita-se assim a animação dos casos de uso, criando uma rede de objectos com ligações (parametrizadas) entre si e simulando o fluxo desde o início do caso de uso até ao fim, de acordo com a informação registada sob a forma de contrato.

4.8 Resumo

Este capítulo apresentou o processo de modelação proposto, salientando o acréscimo da importância da fase de análise, consubstanciada na correcta identificação e descrição dos casos de uso como mecanismo prioritário para a descoberta e validação dos requisitos.

A metodologia proposta apresenta uma sequência de passos que permitem capturar a informação necessária para a captura dos requisitos, a descrição dos cenários associados a um caso de uso, a descrição das actividades que decorrem de um caso de uso e das tarefas associadas. Complementar ao processo iterativo e multi-vista que constitui a fase de análise, apresentou-se também o processo de incorporação de rigor na descrição de casos de uso, de forma a recolher a informação implícita que concerne às restrições à execução de um caso de uso, nomeadamente à definição das pré e pós-condições e do invariante. A introdução de informação rigorosa nestas restrições conjuntamente com a especificação dos fluxos de execução possíveis, permite a prova de que as condições impostas são respeitadas e possibilita a determinação do resultado do caso de uso.

Apresentou-se uma proposta de processo de modelação com utilização da UML para a fase de análise de um sistema, permitindo que se incorpore mais informação que a usualmente recolhida. A proposta aumenta o grau de rigor da descrição dos casos de

uso e torna-os o elemento central da análise de um sistema. A proposta de modelação traçada apresenta três vectores que a justificam: i) o reforço da formalidade dos casos de uso por inclusão de informação rigorosa, ii) a construção de uma abordagem unificada com recurso a várias vistas do mesmo sistema, como mecanismo de consolidação semântica da modelação e iii) uma preocupação em operacionalizar o processo através do recurso a prototipagem.

No que concerne à especificação dos casos de uso, a utilização de diagramas de estado mostra-se importante para a descrição do comportamento exibido. Este passo de especificação permitiu também introduzir mais cedo no processo de análise os aspectos comportamentais, assegurando um conhecimento mais alargado do domínio do problema e da aplicação.

Capítulo 5

Camada de prototipagem de requisitos

5.1 Introdução

No capítulo anterior ao apresentar-se o processo de modelação proposto, atribuiu-se uma maior importância à fase de análise e às tarefas de recolha de requisitos e consequente captura do domínio da aplicação. O acréscimo de importância dado pelo processo a essas tarefas, resulta em documentos de especificação mais elaborados e com maior poder descritivo. Resulta também do processo um detalhe exaustivo dos casos de uso e a identificação dos fios de execução passíveis de serem observados a partir da invocação do mesmo. Os diagramas de estado e sequência criados para detalhar cada caso de uso, possuem uma componente que pode ser facilmente operacionalizada permitindo que se faça, ainda nesta fase de análise, uma validação operacional do modelo.

Além disso, a introdução de rigor no processo de detalhe dos casos de uso, permite levantar ainda mais informação sobre as operações invocadas, suas condicionantes e resultados esperados. Com toda esta capacidade descritiva aumentada, é possível à equipa de projecto, validar muito rapidamente a informação constante do modelo e simular a execução do caso de uso, possibilitando a recolha imediata da reacção do cliente.

Obtém-se desta forma, uma mais rápida realimentação do processo de modelação, na medida em que é possível validar, ainda durante a fase de análise, o modelo que está a ser construído. A não existência de validação nesta fase pressupõe que eventuais

falhas de modelação durante a fase de análise, serão apenas detectadas mais tarde já numa fase de desenvolvimento do sistema.

A validação operacional a realizar deve prever a emulação de comportamentos que os sistemas a serem modelados possam descrever. Na medida em que se aborda a temática do desenvolvimento de sistemas software deve ser prevista a capacidade de prototipagem de tarefas complexas com recurso a diálogos com concorrência e com constrangimentos temporais, de forma a existir efectiva validação dos modelos.

Este capítulo apresenta o suporte criado para o estabelecimento da camada de validação operacional dos modelos. Esta é concretizada numa arquitectura desenvolvida em JAVA e composta por objectos activos interligados por canais de comunicação que implementam diversos protocolos de comportamento. Aborda-se também o papel do protótipo no processo de projecto e a forma como se perspectiva a sua possível reutilização na fase de desenvolvimento. Por fim, apresentam-se as transformações que devem ser feitas para descrever os diagramas de estados em código que utilize a plataforma.

5.2 Suporte tecnológico da plataforma

A introdução da capacidade de um objecto JAVA ter mais do que um fio de execução (*thread*), e de tal capacidade estar incorporada nativamente na linguagem, torna JAVA uma linguagem candidata ao suporte de arquitecturas com o propósito enunciado. O facto de poder responder com relativa facilidade à criação de *threads*, e a capacidade de ser uma linguagem com um eficiente suporte à distribuição, permitem alargar o conceito de sistemas de entidades locais (e correndo no mesmo processo), para sistemas em que nem todos os objectos são locais, podendo inclusive residir em máquinas de arquitecturas diferentes. Além do suporte existente à capacidade de gerir eficazmente *threads*, JAVA apresenta-se também como uma boa solução para a criação de uma arquitectura de suporte à validação dos modelos da fase de análise por ser uma linguagem orientada aos objectos, permitindo-se o desenvolvimento incremental da plataforma recorrendo aos conhecidos métodos de generalização e especialização.

As notações de modelação por objectos, como UML, permitem a descrição de interacção entre os objectos, detalhando se os mesmos são ou não activos e descrevem a forma como as mensagens são enviadas entre eles, bem como o tipo de comunicação associado às mensagens.

Os tipos de mensagens que importa descrever são:

- síncronas;
- perfeitamente assíncronas;
- com um mecanismo de *timeout*;
- métodos que testam condições, isto é, que verificam um invariante de estado antes de se comprometerem com a comunicação.

Não é pois de estranhar que tendo os modelos de análise por objectos este tipo de preocupações e este nível de detalhe, as arquitecturas propostas para estes problemas se baseiem em linguagens por objectos. Adicionalmente, estas linguagens têm evoluído de forma a tornar mais eficiente e transparente o processo de comunicação entre máquinas e processos. Além disso, JAVA, bem como as demais linguagens por objectos, devido à própria essência do paradigma dos objectos, permite que durante a fase de prototipagem o engenheiro de software se centre apenas no estado interno e comportamento dos objectos que está a descrever, abstraindo-se do restante ambiente. A programação é assim naturalmente incremental e sujeita a passos de refinamento sucessivos.

O suporte de JAVA à programação concorrente [Andrews 91] baseia-se em três factores principais:

1. A existência de suporte, dada pela classe `java.lang.Thread` de que as classes do utilizador podem ser subclasses, bastando para isso modificar o método `run()` para definir o seu comportamento;
2. A existência dos prefixos `synchronized` e `volatile` para classificar as variáveis que são acedidas por mais do que uma *thread*;
3. Os métodos `wait`, `notify` e `notifyAll` para coordenar actividades entre as diferentes *threads*.

Todo o método que seja declarado como `synchronized` pode invocar um `wait`, o que provoca que toda a actividade da *thread* em que o `wait` ocorre seja suspensa. Todas as *threads* que estejam paradas no contexto de um objecto podem ser activadas de novo se o método `notifyAll()` for invocado. O método `notify()` apenas activa uma *thread*, permitindo-se assim ao programador um maior controlo sobre os fios de execução a bloquear. Os métodos `wait()`, `notify` e `notifyAll` são normalmente designados por métodos monitores, dado que permitem implementar o conceito de *monitor*, introduzido pela primeira vez por Hoare [Hoare 74], isto é, algo que controla o acesso simultâneo

de várias *threads* a um mesmo objecto. É usual que estes métodos sejam invocados em métodos que funcionam como *guardas*, ou seja, que determinam se o estado do objecto que vai ser acedido é coerente, permitindo esse mesmo acesso ou então bloqueando-o.

5.3 Comunicação entre objectos

A execução de prototipagem sobre os casos de uso implica que do ponto de vista da plataforma de prototipagem de suporte se criem objectos para representar os diversos intervenientes na execução.

Esses objectos podem inclusive conter, ou resultar da composição de, objectos remotos (noutras máquinas), ou então exteriores ao processo JAVA[1]. É pois necessário encontrar um mecanismo que permita a estes objectos interagirem, e mutuamente, trocarem valores. Esse ambiente de comunicação deverá preservar as características de salvaguarda da coerência do estado interno dos objectos acedidos por várias *threads*.

Tal ambiente pode ser baseado em várias soluções, que vão desde a utilização de memória partilhada, recorrendo a primitivas de Inter Process Communication (IPC), a utilização de *streams* para assegurar comunicação ou a outras soluções como sejam o CORBA ou o mecanismo de Remote Method Invocation (RMI) existente em JAVA. Referem-se de seguida algumas destas alternativas.

5.3.1 Memória Partilhada

O recurso a memória partilhada para efectuar a troca de mensagens (ou valores) é uma solução bastante eficaz, e utilizada quando as aplicações são constituídas por apenas um fio de execução. O problema complica-se quando existe mais do que um processo envolvido no sistema, tendo espaços de endereçamento distintos. Soluções existentes para este tipo de problemas passam pela criação de um ambiente em que se lineariza a memória de vários processos e máquinas, de modo a que a aplicação julgue que existe um espaço de endereçamento único. Este tipo de soluções é, por exemplo, bastante utilizado em aplicações que tiram partido de sistemas que permitem visualizar um espaço de memória virtual inter-máquinas. A programação torna-se assim transparente e não reflecte a existência de sistemas, ou processos, remotos. O único tipo de cuidado que deve ser tomado é garantir que o acesso a zonas de memória

[1] O que é um caso particular de objecto remoto.

partilhada é mutuamente exclusivo (quando as operações que a elas acedem podem modificar os dados).

5.3.2 Envio de Mensagens

Nas linguagens por objectos não há usualmente necessidade de recorrer a declaração de memória partilhada, porque é possível a um objecto enviar directamente uma mensagem a outro. Isto é, os objectos fazem parte do mesmo processo, são conhecidos os apontadores para eles e é fácil o envio da mensagem. As mensagens podem ser parametrizadas de modo a serem contentores de valores. JAVA suporta o envio de mensagens entre *threads* diferentes, através da utilização de métodos `synchronized`, com o recurso a `wait` e `notify`. O tradicional envio de mensagens não constitui no entanto uma solução válida quando nem todos os objectos são locais, isto é, não pertencem ao mesmo programa JAVA. A comunicação entre objectos necessita, nestas circunstâncias, de uma interface que efectue a ligação entre os objectos. Este problema pode ser evitado utilizando a potencialidade que a linguagem oferece de invocação de métodos em objectos remotos, o RMI. O *Remote Method Invocation* facilita a implementação de uma arquitectura distribuída de objectos, fornecendo para isso um modelo com elevado nível de abstracção. O RMI é uma abordagem semelhante ao Remote Procedure Call (RPC), mas que tira partido de ser inerentemente orientado aos objectos. Nesta abordagem permite-se enviar objectos como parâmetros dos métodos a invocar remotamente, sendo a única restrição existente que esses objectos sejam *serializáveis*, isto é, que implementem a interface `Serializable`, o que significa poderem ser guardados em memória secundária.

5.3.3 Streams de Input e Output

A comunicação entre objectos, quando não é possível entre estes efectuar a usual troca de mensagens, pode ser realizada através de *Streams*. Streams são meios de comunicação que funcionam como interface de Input/Output para memória, para o sistema de ficheiros, para ligações de rede, etc. Caracterizam-se por serem estruturas de dados perfeitamente assíncronas, que têm pontos forçados de sincronismo quando se encontram vazias ou cheias. JAVA fornece uma quantidade apreciável de Streams, sendo particularmente úteis as ObjectInputStreams e ObjectOutputStreams pois permitem um uso transparente e asseguram de modo eficaz a persistência.

5.3.4 Canais

Ao invés das soluções para a comunicação entre objectos vistas nos pontos anteriores, o conceito de canal não é um conceito comum nas linguagens de programação por objectos. O conceito e declaração de canais é usual em sistemas de prototipagem de linguagens baseadas no CSP [Hoare 85], e na linguagem OCCAM [Jones 88], que é a concretização da parte comportamental do CSP e era a linguagem de programação de sistemas paralelos normalmente baseados em arquitecturas de *transputers*.

O canal acaba assim por ser um objecto que contém dados que são conceptualmente partilhados pelos detentores do canal, correspondendo a uma utilização de *memória partilhada*, e cujo acesso pode eventualmente ser feito através de *Streams*, sendo que os canais para ligar objectos, ou entidades, em máquinas diferentes recorrem efectivamente ao uso destes. Apresenta-se de seguida a estrutura da arquitectura proposta para facilitar a validação operacional do diálogo existente na execução dos diversos cenários de um caso de uso. Esta arquitectura assenta fundamentalmente na existência da entidade Canal como mecanismo estruturante para a prototipagem [Ribeiro 98].

5.4 Arquitectura de Suporte

O conjunto de componentes JAVA a seguir descrito implementa as entidades necessárias para o suporte eficiente e simples do processo de prototipagem associado à fase de análise.

Fornecendo este conjunto de classes, e também um método de utilização, a codificação das entidades descritas na linguagem apresentada torna-se mais transparente e permite um maior nível de abstracção a quem a utiliza. Pretende-se assim diminuir os erros típicos das aplicações com diálogos complexos, concorrentes ou assentes em restrições temporais, e fornecer uma funcionalidade básica, que pode ser até refinada, para o suporte eficiente de aplicações com estas características.

5.4.1 A entidade Canal

O conceito básico da arquitectura assenta na definição de Canal, entidade através da qual se trocam valores entre objectos, valores esses que podem ser tipos de dados, ou então, mais genericamente, invocação de eventos. O canal deve assim providenciar a capacidade de nele se escreverem, e lerem valores. Sendo o canal partilhado por

entidades que fazem parte de *threads* diferentes (ou até de processos diferentes em máquinas também distintas) é necessário garantir que o acesso aos dados do canal é atómico[2].

O canal deve oferecer a funcionalidade básica para o tratamento das expressões de escrita de valores no canal e de leitura de valores desse mesmo canal.

A classe `Canal` garante a atomicidade e a não interrupção das operações de leitura e escrita, através de duas variáveis que funcionam como semáforos. A declaração de duas variáveis, `semaforo_leitura` e `semaforo_escrita`, evita que se possam efectuar operações em simultâneo.

```
public class Canal {
  ...
  ...
  //objectos que controlam se podemos aceder aos objectos
  //para escrita e/ou leitura
  private Object semaforo_escrita = new Object();
  private Object semaforo_leitura = new Object();
  ...
}
```

No caso de conceptualmente se pretender ler e escrever no mesmo canal, é necessário prever a declaração de dois canais, um destinado a escrita e outro a leitura. Isto corresponde a ter duas vistas sobre um canal, uma em que apenas é visível a operação de escrita e outra em que apenas se tem acesso à operação de leitura. Em JAVA não se torna necessário criar duas classes distintas, cada uma com uma operação, bastando para tal declarar esses comportamentos distintos como sendo compatíveis com uma determinada interface JAVA. As interfaces `CanalLeitura` e `CanalEscrita` que se apresentam de seguida, foram criadas com base nesse pressuposto.

`CanalLeitura` é uma interface que providencia o metodo de leitura de um canal, sendo que quem se declarar como sendo deste tipo apenas tem capacidade de leitura. A sua descrição é:

```
public interface CanalLeitura {

  public StandardProtocol read()
                          throws ExcepcaoLeituraCanal, IOException;

}
```

[2]A atomicidade não é garantida a atomicidade nas operações para objectos definidos pelo utilizador.

`CanalEscrita` apenas disponibiliza o método de escrita e tem como definição:

```
public interface CanalEscrita {

  public boolean write(StandardProtocol object)
                throws ExcepcaoEscritaCanal, IOException;

}
```

Quem utiliza a arquitectura proposta pode detalhar se pretende declarar instâncias de `Canal`, ou de `CanalLeitura` e de `CanalEscrita`. Conforme a semântica das interfaces JAVA, a uma entidade do tipo `CanalLeitura` não podem ser enviados outros métodos que não `read()`. A declaração de canais obedece assim ao seguinte protocolo,

```
CanalLeitura canalin  = new Canal();
CanalEscrita canalout = new Canal();
```

Existem diversos tipos de composição entre objectos envolvidos em diálogos. Esses tipos de composição são: a composição sequencial, alternativa e paralela. Assume-se que ao criar um canal usando o construtor por omissão, este será utilizado em construções sequenciais. Se não for este o caso, deverá ser explicitamente especificado o tipo de construção pretendida, tal como se detalhará numa próxima secção.

Quando um objecto envia uma mensagem a outro o comportamento esperado é o de sincronismo, isto é, o objecto que envia a mensagem fica à espera da resposta enviada pelo receptor. É possível elaborar mais e definir ainda outros tipos de sincronização existentes, por exemplo, baseados em *buffers* ou guardas. A arquitectura apresenta alguns destes tipos de sincronização, deixando contudo a possibilidade de o utilizador criar os seus próprios ambientes de sincronização. Quando nada for especificado, a sincronização assumida será a sincronização absoluta[3]. Quer isto dizer que um *"escritor"* só pode voltar a enviar dados para o canal quando alguém já os tiver consumido (lido), e que um *"leitor"* não lê duas vezes os mesmos dados.

O canal, como entidade estruturante da arquitectura que coordena a passagem de valores entre dois objectos, é também o principal garante de uma arquitectura suficientemente aberta, de modo a que um qualquer utilizador possa criar canais com um comportamento por si definido. A entidade canal funciona assim como uma entidade agregadora de componentes que determinam o seu funcionamento. O canal agrega entidades que determinam o tipo de sincronização pretendido e a composição (sequencial, alternativa ou paralela) entre os intervenientes.

[3]Também referenciada na literatura da área como *Rendez-vous*.

A classe `Canal` disponibiliza, para além do construtor por omissão, `Canal()`, outros construtores aonde é possível dizer qual o tipo de sincronização pretendida. Por exemplo, o construtor `Canal(Sincronizacao tiposinc)` permite que seja criado um canal com um tipo de sincronização `tiposinc`, se esta for existente e disponível na arquitectura.

O método read()

O método de leitura do canal deve garantir que não existe concorrência interna no objecto canal, evitando assim que o estado interno deste possa ser tornado incoerente. O tipo de controlo exercido na comunicação entre as entidades que usam o canal não é tratado a este nível, uma vez que depende do tipo de sincronização pretendido. Sendo assim, este `read()`, depois de verificar que conseguiu acesso de execução, apenas invoca o `read()` que a classe que implementa a sincronização definiu.

O método deve respeitar três níveis de concorrência interna para se poder invocar:

1. garantir que ninguém mais tem acesso à variável `semaforo_leitura`;
2. garantir que nenhum outro método é invocado no objecto canal;
3. verificar o estado de sincronização antes de invocar o método de leitura; se não o conseguir fazer, a *thread* deve então bloquear.

Quando o objecto tiver lido os dados do canal, deve notificar eventuais entidades envolvidas e que tenham ficado bloqueadas à espera de efectuar uma leitura. Deve nesses casos ser enviado um `notify`.

Apresenta-se de seguida o template do método `read()` da classe `Canal`.

```
public synchronized StandardProtocol read() throws ExcepcaoLeituraCanal,
                                                    Exception {
  StandardProtocol object;

  synchronized (semaforo_leitura) {
    synchronized (this) {
      if (tiposinc.get_estado() == Sincronizacao.EMPTY)
        wait();   //se nao houver nada para ler, esperar
      object = tiposinc.read();
      notify();   //avisar quem estiver bloqueado
    }
    return object;
```

```
  }
}
```

A primitiva `synchronized` estabelece uma fila de espera para quem tente entrar num dos níveis de segurança. Quando uma das *threads* acaba a sua tarefa, é escalonada a primeira posição da fila de espera. A constante `EMPTY` indica que não existe nada para ler no canal.

O arquétipo apresentado é válido para as situações em que a composição é feita sequencialmente e, assim, em cada instante só dedica atenção a um canal.

O método write()

O método `write()` é muito similar no funcionamento ao atrás apresentado, sendo que agora o agente que escreve deve testar se a pré-condição do canal é respeitada, isto é, se é possível escrever.

```
public synchronized boolean write(StandardProtocol object)
                            throws ExcepcaoEscritaCanal, Exception {
  boolean resultado = false; //valor inicial

  synchronized (semaforo_escrita) {
    synchronized (this) {
      if (tiposinc.get_estado() == Sincronizacao.FULL)
        wait();   //Esperar para garantir sincronizacao
      resultado = tiposinc.write(object);
      notify();   //avisar quem estiver bloqueado
    }
    return resultado;
  }
}
```

5.4.2 A Interface StandardProtocol

Os valores a serem enviados para os canais devem obedecer a critérios que possibilitem uma correcta implementação da descrição feita. Para garantir que não existe partilha de referências entre objectos que são enviados por um canal, este apenas permite que sejam transmitidas réplicas desses objectos.

Além deste aspecto, todos os objectos que passam num canal devem poder ser comparados. Ora estas operações são da competência de cada objecto, e devem ser

obrigatoriamente implementadas pela classe a que o objecto pertence[4]. Esta não é, aliás, uma característica metodológica dos objectos enviados pelos canais, mas sim uma característica que todos os objectos JAVA deveriam ter.

Os objectos a enviar pelos canais devem ser do tipo `StandardProtocol`, interface que define os seguintes métodos:

```
public interface StandardProtocol extends Serializable, Cloneable
{
  //Metodo que permite replicar um objecto
  public Object clone();

  //Metodo que permite comparar dois objectos
  public boolean equals(StandardProtocol anObject);

  //Visualizar o objecto sobre a forma de String
  public String toString();
}
```

Conforme se apresentou na descrição dos métodos `read()` e `write()`, os objectos lidos e escritos são do tipo `StandardProtocol`, o que significa que só quem implementar esta interface é candidato a ser utilizado na arquitectura.

A arquitectura de suporte fornece as classes para os tipos de dados básicos, sendo o utilizador livre para implementar dados com estrutura mais complexa, desde que para estes defina os métodos `clone()`, `equals()` e `toString()`.

O suporte fornecido para os valores inteiros corresponde à seguinte classe:

```
public class IntStandard implements StandardProtocol {
  int conteudo;
  //construtores
  public IntStandard() {
  }
  public IntStandard(int valor) {
    this.conteudo = valor;
  }
  int get_conteudo() {
    return this.conteudo;
  }
  //Metodos de StandardProtocol
  public Object clone() {
```

[4]O suporte implícito que o JAVA fornece para este tipo de operações é insuficiente.

```
    return (Object)(new IntStandard(this.conteudo));
  }
  public boolean equals(StandardProtocol object) {
    if ((object != null) && (object.getClass() == this.getClass()))
      return(((IntStandard)object).get_conteudo() == this.conteudo);
    else
      return false;
  }
  public String toString() {
    return((new Integer(conteudo)).toString());
  }
}
```

A hierarquia dos tipos de dados pode ser vista na Figura 5.1.

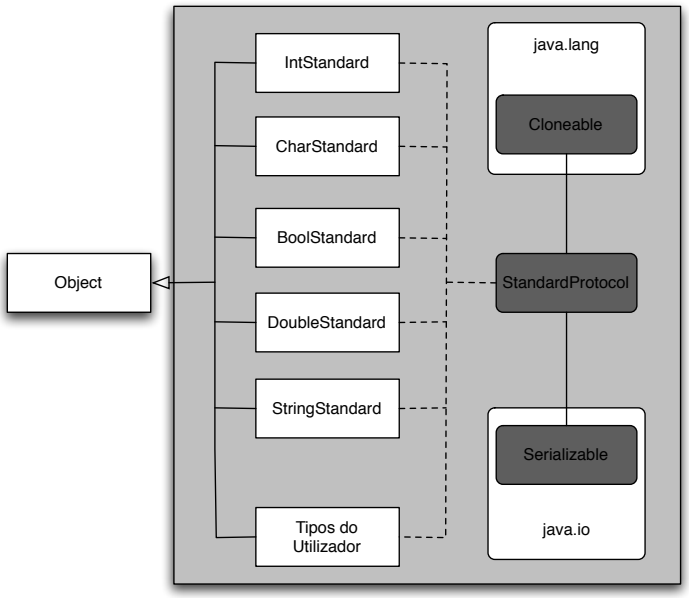


Figura 5.1: Hierarquia de Tipos de Dados.

5.4.3 Tipos de Sincronização

Definidas que estão as operações de um canal e apresentados os tipos de dados que circulam nos canais, é agora necessário falar na sincronização existente entre as entidades que o partilham. Este é mesmo um dos aspectos mais importantes da arquitectura, uma

vez que permite a interligação de diferentes objectos com diversos tipos de comunicação entre eles, criando um nível de diálogo mais rico.

Podem desde já identificar-se alguns tipos de sincronizações habituais e que são utilizadas na maior parte das linguagens. Apesar de se fornecer um leque de classes de sincronização suficiente para implementar a maior parte dos problemas, qualquer utilizador poderá estender a classe `Sincronização` e codificar um protocolo próprio.

A classe `Sincronização` é uma classe abstracta que agrega o(s) objecto(s) que os escritores e leitores escrevem ou lêem. Esse objecto, ou agregação de objectos, pode ser uma posição de memória, de disco, uma ligação de rede, etc. A classe `Sincronização` disponibiliza para o acesso aos dados uma variável, `estado`, que pode ter um de três valores:

1. EMPTY - não existe nada para ler, o que significa que os possíveis processos leitor vão bloquear. Quem quiser escrever no canal tem acesso ao mesmo;

2. FULL - processos que queiram ler podem fazê-lo. Os possíveis escritores devem ser bloqueados;

3. NOCONSTRAINT - não existe nenhuma restrição nem à leitura nem à escrita. Quando este for o estado do canal, o comportamento do canal é assíncrono.

A classe `Sincronização`, além da variável que determina se quem escreve ou lê o pode fazer, disponibiliza também a assinatura dos métodos que permitem efectuar o `read()` e `write()` no canal. Como a implementação destes métodos depende do tipo de comportamento pretendido, não são a este nível de facto implementados, ou seja, são métodos abstractos, delegando essa responsabilidade nas subclasses.

```
public abstract StandardProtocol read()
     throws  ExcepcaoLeituraCanal, Exception;

public abstract boolean write(StandardProtocol object)
     throws  ExcepcaoEscritaCanal, Exception;
```

A hierarquia actual de subclasses de `Sincronização` é a apresentada na Figura 5.2.

A arquitectura fornece ao utilizador, além das suas próprias definições, os tipos de sincronização que se referem:

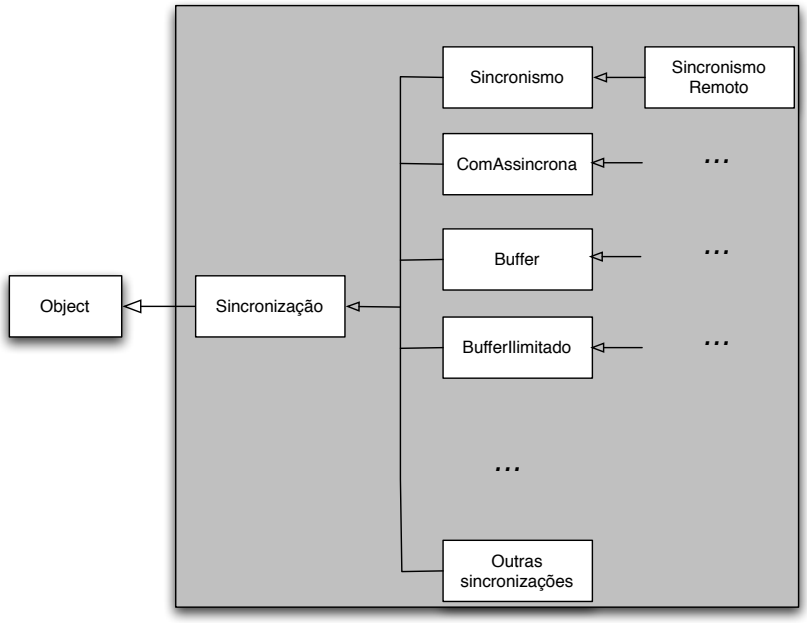


Figura 5.2: Hierarquia de Classes de Sincronização.

- Sincronismo Absoluto - sincronização absoluta entre leitores e escritores de dados no canal. A uma escrita sucede-se uma leitura, seguida de nova escrita e assim sucessivamente. Essa preocupação é evidenciada no código apresentado a seguir em que se verifica a utilização de `EMPTY` e `FULL` para garantir a sincronização.

```
public synchronized StandardProtocol read() throws ExcepcaoLeituraCanal {
    StandardProtocol objecto; //local para onde se le

    objecto = (StandardProtocol)(object.clone());
    this.set_estado(this.EMPTY);   //quem quiser pode escrever
    return objecto;
  }

  public synchronized boolean write(StandardProtocol objecto)
                              throws ExcepcaoEscritaCanal {
    object = (StandardProtocol)(objecto.clone());
    this.set_estado(this.FULL); //quem quiser pode ler
    return true;
  }
```

- Comunicação Assíncrona - comunicação perfeitamente assíncrona entre escritores

e leitores. O processo escritor não espera que o leitor consuma os dados, e pode escrever sempre que ninguém esteja a aceder concorrentemente ao canal. Quando um leitor efectua uma leitura, consome o valor.

Note-se que este tipo de comportamento assíncrono tem no entanto laivos de sincronismo quando alguém pretender ler dados e ainda ninguém os escreveu. Nesta situação os leitores bloqueiam, visto que esperam que o valor da variável de instância `estado` passe a `NOCONSTRAINT`;

Neste tipo de sincronização, quando o valor é `EMPTY` significa que foi consumido o valor que tinha sido colocado no canal. A implementação de `read()` e `write()` no caso da comunicação assíncrona é apresentada de seguida.

```
public StandardProtocol read() throws ExcepcaoLeituraCanal {
  StandardProtocol objecto = (StandardProtocol)(this.object.clone());
  this.set_estado(EMPTY); //para consumir o valor
  return objecto;
}

public boolean write(StandardProtocol object)
               throws ExcepcaoEscritaCanal {
  this.object = (StandardProtocol)object.clone();
  this.set_estado(NOCONSTRAINT); //para evitar que os escritores e
                                    //leitores façam wait()
  return true;
}
```

- Buffer Limitado e Ilimitado - Estes dois tipos de ambientes de sincronização são a generalização dos dois apresentados anteriormente. Repare-se que tanto o ambiente de sincronismo absoluto como o de comunicação assíncrona são casos particulares em que o *buffer* só tem uma posição. Ambientes de sincronização com mais do que uma posição para guardar valores são mais propícios a comportamentos assíncronos[5].

  O *buffer* ilimitado tem apenas um ponto de sincronismo forçado, que ocorre quando alguém tenta ler algo a partir de um estado vazio. No desenvolvimento da arquitectura utilizou-se um array dinâmico, no caso do tipo `ArrayList` na medida em que instâncias desta classe podem aumentar de tamanho em tempo de execução, e também porque possuem funcionalidade que permite serem tratadas como filas de espera.

[5]Veja-se por analogia os *buffers* das mailboxes. Quem envia um mail não fica bloqueado à espera que este seja lido.

A implementação de um ambiente de sincronização com um buffer limitado implica um certo cuidado na sua utilização, uma vez que podem ocorrer situações de bloqueio (*deadlock*) derivadas da tentativa de ler quando não existem dados no *buffer*, ou então, de tentar escrever e o *buffer* estar cheio. O *buffer* com tamanho pré-definido deve ser tratado como um array circular, de modo a optimizar o seu funcionamento, evitando assim estar sempre a efectuar deslocamentos dos elementos quando um valor é consumido.

Note-se que nos canais implementados segundo a noção de *buffer*, limitado ou não, deve ser garantido que todos os objectos pertencem ao mesmo tipo de dados. É necessário passar como parâmetro no caso do BufferIlimitado, o tipo dos dados no construtor do canal para posteriormente poder testar se o que se quer escrever é do tipo correcto.

Tal como a hierarquia de tipos de dados apresentada, também a hierarquia de classes de sincronização pode ser facilmente aumentada. O utilizador só deve conhecer a assinatura dos métodos abstractos declarados em Sincronização.

5.4.4 Composições de Alternativa e Paralelismo

Quando uma entidade está a comunicar com outro(s), recebendo e enviando valores, nem sempre é necessário assegurar que existe uma ordem sequencial nas operações que utilizam o canal. A ordem sequencial é necessária quando as instruções que se seguem a uma operação feita num canal (`read()` ou `write()`) necessitam de conhecer os dados que foram trocados. Veja-se o exemplo da Figura 5.3, em que o objecto A recebe dados via o canal $canal_1$, e depois de efectuar uma qualquer operação sobre esses dados, envia o resultado para o objecto B. O objecto B comporta-se exactamente como A e envia os resultados para C, que devolve novamente o resultado da aplicação das operações.

Este tipo de comportamento é naturalmente sequencial, isto é, qualquer das entidades fica à espera de ter dados disponíveis para processar. Quando os obtém, transforma-os e envia-os para o próximo elemento da cadeia.

Este comportamento segue o seguinte algoritmo:

```
StandardProtocol v = canal_i.read();
canal_i.write(f(v));
StandardProtocol v = canal_i.read();
```

Este comportamento não permite qualquer tipo de alternativa que permita efectuar mais do que uma operação sobre os canais *"ao mesmo tempo"*. Na arquitectura,

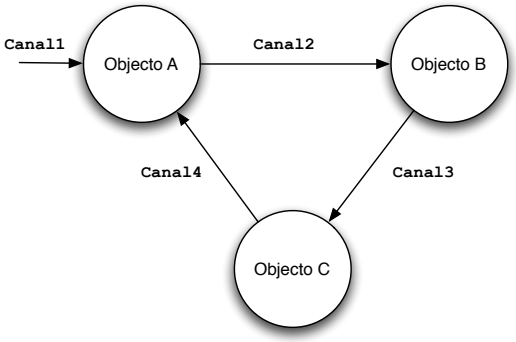


Figura 5.3: Execução Sequencial.

quem utiliza os `read()` e `write()` sem especificar mais nada, utiliza-os com um cariz nitidamente sequencial.

Uma situação em que o comportamento sequencial não é o único a poder ser utilizado corresponde à leitura ou escrita em *simultâneo*. Veja-se o objecto apresentado na Figura 5.4, que recebe dados via o canal_0 e depois escreve para os n outros objectos apresentados o resultado de efectuar uma qualquer operação sobre esses dados. Este tipo de comportamento pode ser descrito de modo sequencial, sendo necessário que o objecto_A estabeleça uma ordem pela qual vai fazendo as sucessivas escritas. Porém, este desempenho pode ser melhorado se for possível executar as n operações de escrita em paralelo, sabendo o utilizador que a ordem pela qual escreve as instruções pode não ser a ordem pela qual elas são efectuadas. É importante também que as diferentes operações candidatas a serem executadas em fios de execução paralelos não interajam entre si. No exemplo apresentado não existe interacção entre as operações, que apenas se limitam a enviar dados para um determinado canal.

Para ter a capacidade de executar diversos fios de execução em simultâneo, é necessário criar um objecto que funcione como entidade que desencadeia várias *threads*. O número de *threads* deverá ser o número de acções que devem decorrer em paralelo. O objecto que implementa o paralelismo necessita de saber quantos fios de execução é que vai controlar. Para esse objecto, que assegura paralelismo, é também necessário passar a acção a efectuar. Quando para cada uma das *threads* for passada uma acção, esta é executada sem haver conhecimento do estado das demais acções.

Definiu-se na arquitectura uma classe designada `Parallel`, cujo construtor necessita de saber o número de *threads* que um objecto desta classe vai executar. Essa necessidade deriva do facto de uma vez criada a instância de `Parallel`, as *threads* se-

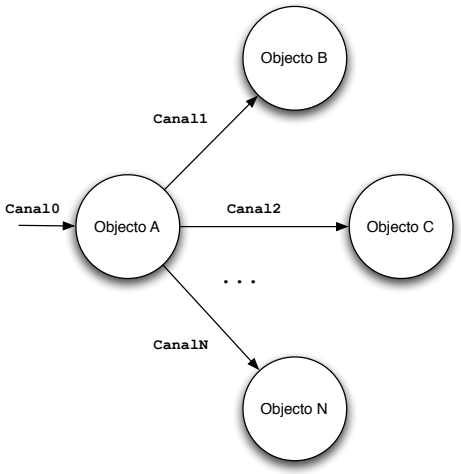


Figura 5.4: Execução Paralela.

rem imediatamente activadas (bloqueiam entretanto até que a acção lhes seja passada). Se o processo de determinação de quantos fios de execução o objecto aloca fosse feito em tempo de execução, haveria uma considerável perda de desempenho.

É também necessário não esquecer que é preciso esperar pela última *thread* a ser executada para abandonar o contexto do composição do tipo `Parallel`.

O algoritmo para a descrição do comportamento exibido na Figura 5.4, utilizando uma instância de `Parallel`, é o seguinte:

```
Parallel bloco_paralelo = new Parallel(N);
StandardProtocol v = canal_0.read();
canal_1.write(bloco_paralelo, f(v));
...
canal_n.write(bloco_paralelo, f(v));
bloco_paralelo.end();
```

Outro tipo de comportamento muito utilizado prende-se com a situação em que um objecto está à espera de valores nos seus vários canais de leitura, e quando recebe dados de um canal segue um determinado fio de execução.

Este comportamento reflecte as ligações de canais visíveis na Figura 5.5.

A modelação deste tipo de comportamento originou a criação de uma entidade, de modo a para abstrair a complexidade destas interacções. Na maioria das utilizações é também garante de o processo não se comprometer em situações que podem levar a bloqueio. Na especificação acima apresentada, se o objecto se compromete numa

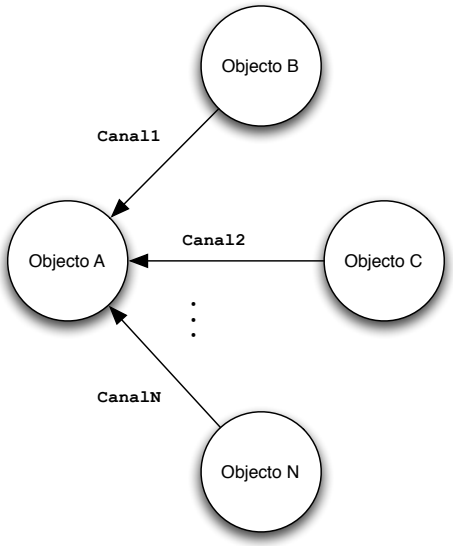


Figura 5.5: Escolha Alternativa.

leitura do canal c_1 e nesse canal nunca se escreve, o comportamento do objecto será inevitavelmente um bloqueio. O objecto *A* deve invocar um método no canal que permita saber o seu estado, e não efectuar um `read()` sem ter a certeza de que o canal onde vai efectuar a leitura tem dados disponíveis. Esse método é enviado a uma instância da classe `Alternativa` que tem a capacidade de devolver um canal no qual seja possível ler. Esse método, que se designa `choice()`, deve ter um mecanismo de espera para evitar que ele próprio cause bloqueio.

O algoritmo para a situação descrita na Figura 5.5, receber valores de uma série de canais sem saber qual o primeiro a estar disponível, é então,

```
Alternativa escolha = new Alternativa();
int i = escolha.choice(canais);
StandardProtocol v = canais[i].read();
```

5.4.5 Comunicação entre agentes remotos

Entre objectos que pertencem ao mesmo processo JAVA, isto é, que foram invocados na mesma sessão da máquina virtual, é possível passar a referência para o apontador do canal partilhado pelos objectos. Isto não é possível quando os objectos ou não pertencem ao mesmo processo JAVA, ou então, residem em máquinas diferentes. A solução de

comunicação, nestes casos, deve ser resolvida por acesso a *streams*, ou então utilizando o suporte da linguagem para estabelecer ligações socket a socket. Relativamente à declaração de um canal, os canais remotos diferem apenas pelo facto de, aquando da sua criação, ser necessário dizer qual a máquina e o porto que constituem as extremidades do canal.

Do ponto de vista de funcionalidade, as classes que implementam ligações remotas continuam a ter o mesmo tipo de controlo na determinação se os escritores ou leitores podem ou não efectuar a sua tarefa. A única mudança prende-se com o acesso aos dados, que tem de ser agora obtido através de instâncias de `ServerSocket`.

5.4.6 Canais de difusão

Num sistema interactivo é em certas circunstâncias necessário enviar mensagens para todos as entidades do sistema, ou para todos aqueles que se tenham prontificado a escutar.

Na arquitectura apresentada, os canais de difusão (*broadcast*) continuam a ser apenas unidireccionais. Do ponto de vista operacional, a difusão de mensagens determina que são efectuadas tantas cópias da mensagem quantos os objectos que estejam ligados ao canal de broadcast como receptores[6].

Um canal de difusão regista os de apontadores para canais do tipo Buffer (limitado ou não), que são pertença dos objectos que participam (como ouvintes) da difusão de dados.

O canal que difunde as mensagens não tem de conhecer qual é o tipo de canal que os receptores estão a utilizar, nem tal faria sentido uma vez que obrigaria a um procedimento de arranque da comunicação bastante mais moroso e complexo. No entanto, deve-se referir que o comportamento do objecto que difunde informação pelo canal de difusão pode ser de sobremaneira influenciado pelos canais dos receptores, dado que pode não conseguir escrever (por falta de espaço, por o receptor ainda não ter consumido os valores, etc[7]) sendo então obrigado a bloquear.

Quando o canal de difusão tem algo para enviar, deve fazê-lo para todos os receptores. O desempenho da difusão pode ser aumentado se este processo for feito no âmbito de um bloco executado em paralelo. O método `write()` dos canais de broadcast utiliza

[6]Este é aliás o comportamento conhecido das listas de distribuição de correio electrónico, que não são mais do que uma implementação de um canal de broadcast.

[7]Depende do tipo de canal e sincronização que o ouvinte disponibilizou.

a possibilidade de escrita em paralelo (disponibilizada por uma instância de `Parallel`) para acelerar o processo de difusão.

A classe `CanalBroadcast` não é subclasse de `Canal`, uma vez que o estado interno de ambas não é compatível, mas reutiliza os conceitos de `Canal` por agregação. Fornece-se nesta classe um método que permite a um objecto juntar-se ao processo de difusão, o método, `add_listener`, que aceita como parâmetro o canal.

5.5 Animação dos Diagramas de Estado

Com o intuito de animar os casos de uso, é necessário utilizar as facilidades que a arquitectura de prototipagem disponibiliza para transformar o diagrama de estado em componentes activos. A transformação dos diagramas de estado em código pode ser razoavelmente complexa se não existir um suporte facilitado para esta tarefa. Isto deve-se aos mecanismos de modelação complexos que são intrínsecos ao modelo dos diagramas de estado. É possível encontrar diversos algoritmos para transformar uma máquina de estados em código fonte, mas a mais comum é aquela que utiliza uma variável de um tipo enumerado (com todas as designações dos estados) que serve de mecanismo de selecção numa estrutura do tipo `switch`. Cada entrada do switch, isto é, cada caso (`case`) é equivalente a um estado do diagrama e cada vez que é efectuada uma transição, a variável que controla o `switch` é actualizada com a identificação do novo estado.

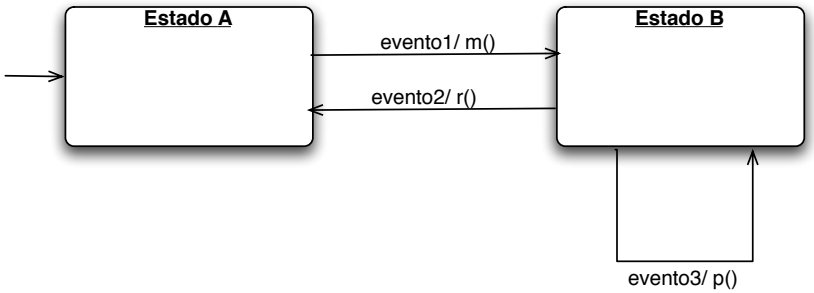


Figura 5.6: Diagrama de estados exemplo.

Uma estratégia como a descrita, faz com que o código que descreve o diagrama de estados da Figura 5.6, seja constituída por um `switch` que é percorrido em função de uma variável que indica qual é o estado actual. Dentro de cada `case` relativo à variável do estado existe um outro `switch` para os eventos que podem ser recebidos.

Esta abordagem é simples e não apresenta especial complexidade na automatização do processo de construção do código a partir do diagrama de estados, mas não é escalável quando aplicada a máquinas de estado mais complexas, previsivelmente com decomposição hierárquica. Parte-se do princípio que a máquina de estados é plana, com todos os estados ao mesmo nível, o que em especificação de comportamentos mais complexos leva a um inevitável aumento do número de estados, e transições, a considerar. Não poderia ser esta a forma escolhida para animar os diagramas de estado, devido a esta limitação na descrição de máquinas de estado complexas. Além disso, a capacidade de introdução no código descrito anteriormente de mecanismos que permitam à equipa de teste mudar, durante o tempo de prototipagem, os valores das variáveis de estado, de forma a simularem alterações ao sistema, impede que a animação possa ser feita desta forma.

É necessário aferir como é que se pode utilizar a arquitectura de suporte para transformar em código fonte o diagrama de estados apresentado. A arquitectura de suporte à prototipagem de requisitos é baseada numa arquitectura de objectos activos interligados entre si, de forma a estabelecer a tipologia pretendida. As interligações são unidireccionais, o que permite descreverem as transições entre estados, e podem ver o seu comportamento especificado de forma a emularem comportamentos síncronos, assíncronos, ou outros.

Numa primeira fase, é necessário que a equipa de analistas, com base no diagrama de estados que se pretende animar, estabeleça a configuração de objectos existentes e inicialize as ligações entre eles. A arquitectura foi idealizada como uma camada de serviço que permite tornar o mais transparente possível a criação de programas que sejam eminentemente reactivos e com necessidades de concorrência na comunicação entre as entidades. Exemplos de sistemas reactivos que são facilmente modelados pela arquitectura são, por exemplo, os sistemas interactivos, cuja camada de apresentação é candidata a ser especificada com recurso aos diagramas de estado, de modo a capturar os eventos que se podem operar sobre a interface.

O algoritmo base para a utilização da arquitectura de suporte parte dos seguintes princípios:

- cada estado é modelado por uma classe que deve exibir como característica base o facto de ser subclasse (mesmo por transitividade) da classe `Thread`, de forma a que se constitua como um objecto activo que espera receber eventos;

- cada objecto, isto é, neste contexto, cada estado, deve estabelecer os canais de interligação com os outros estados de acordo com a especificação evidenciada pelo

diagrama. Esta é uma operação que é feita aquando da criação do objecto estado, e as ligações são passadas como parâmetros do construtor. Cada um dos objectos é responsável por marcar cada uma das interligações como sendo de leitura, no caso em que corresponde a uma transição para o estado em causa, ou como sendo de escrita, nas situações em que são transições originadas no estado. Na suporte fornecido pela arquitectura existem duas interfaces, CanalLeitura e CanalEscrita que permitem classificar cada uma das interligações.

No caso do diagrama de estado da Figura 5.6 a classe EstadoA tem como transição de saída a ligação etiquetada como evento1 e como transição de entrada a ligação que está etiquetada como evento2. Em qualquer um das transições, de saída e de entrada, elas implicam a execução das acções `m()` e `r()`, respectivamente, e

- em cada um dos objectos, redefinir o método `run()`, de forma a descrever as expressões que regulam a actividade que se executa dentro de cada estado. No caso do EstadoA que estamos a descrever, não é necessário fazer mais do que contemplar a recepção de eventos (neste caso evento1), de forma a originar a transição e invocar a acção (`m()`) associada à mesma.

Os diagramas de estado são especialmente relevantes para aferir da correcta especificação do caso de uso, logo para validar a captura de requisitos, em situações em que o sistema software seja reactivo. Nessas situações, o sistema reage em função da história, dos valores do seu estado e também da invocação não previsível de acções externas sobre o sistema. No diagrama de estados que foi apresentado, se quando o caso de uso estiver no EstadoA não for enviado o evento evento1, não existe nenhuma transição e nada se altera no sistema. Ora, quando o sistema estiver em utilização, a chegada desse evento é determinada pela vontade de quem o está a utilizar, pelo que na fase de prototipagem também é importante que o seu envio possa ser controlado por quem está a testar (validar) os requisitos.

Para que seja possível desencadear eventos nos objectos que representam os estados, a solução que a arquitectura de suporte disponibiliza baseia-se na utilização de canais de controlo que são agrupados num único objecto que permite a comunicação com os objectos que constituem a representação do diagrama de estado. Tipicamente para estabelecer este nível de controlo, podem ser utilizados os mesmos tipos de canais que foram acima identificados para as transições, não sendo necessário especificar um novo tipo de dados para a comunicação de valores, na medida em que o que é determinante é o facto de se poder discriminar qual é a transição que se pretende activar.

A estratégia de concretização em código de um diagrama de estados, pode ser

baseada numa solução arquitectural em que a classe que representa um estado é parametrizada pelas características que um estado deve possuir. A informação que é comum a todos os estados de uma máquina de estados é a que se apresenta:

- um conjunto de transições que tem como origem o estado e como destino um outro estado do diagrama. Para cada uma dessas transições estão definidos o evento, a condição e a acção;
- um conjunto de acções associadas às actividades internas do estado, por exemplo entry, exit, do, entre outras.

Para generalizar o conceito de acção que corresponde a uma operação que é executada num determinado contexto, definiu-se uma interface, `Executor`, que possui apenas a assinatura do método `execute`. A definição de `Executor` é a seguinte:

```
public interface Executor {
  public void execute();
}
```

Com base nesta definição qualquer algoritmo que pretenda ser executado no contexto de um estado apenas tem de codificar o método `execute`. Evita-se recorrendo a esta estratégia arquitectural que para representar um diagrama de estado se tenham que criar objectos distintos para cada um dos estados, quando a diferença entre eles é apenas a composição do seu estado interno (eventos, ligações e acções). A vantagem acrescida de colocar as operações como sendo objectos deriva do facto de ser possível, desta forma, reutilizar algoritmos pois basta instanciar novamente o objecto com a definição do método `execute`.

A definição de `Executor` é também especialmente interessante do ponto de vista de transformação das declarações OCL em código. Cada um dos comandos do tipo `Executor` contém uma expressão derivada da expressão OCL e quando o estado é activado e invoca o método `execute`, esse código é invocado.

Por vezes a invocação de uma acção está dependente da avaliação de uma condição. Nas transições quando não se escreve a condição é porque o valor lógico dela é verdadeiro. Nas transições de estado, um evento está associado a um par condição/acção, isto é, se a condição for verdadeira a acção é executada. No caso de não existir nenhuma acção associada à transição isso corresponde a um `Executor` com uma definição vazia. Da mesma forma quando não estiver prevista uma condição, esta pode ser considerada como `true`, permitindo dessa forma que a acção seja executada.

Define-se a classe `ParCondAccao` que agrega duas variáveis de instância e tem a definição seguinte:

```
public class ParCondAccao {
  boolean  condicao;
  Executor accao;

  ...
}
```

Para descrever as actividades internas do estado apenas é necessário associar o tipo de actividade ao comportamento que se pretende ter. Dessa forma o comportamento interno de um estado é um mapeamento da actividade para objectos do tipo `Executor`.

A Figura 5.7 apresenta a definição arquitectural da classe `Estado`.

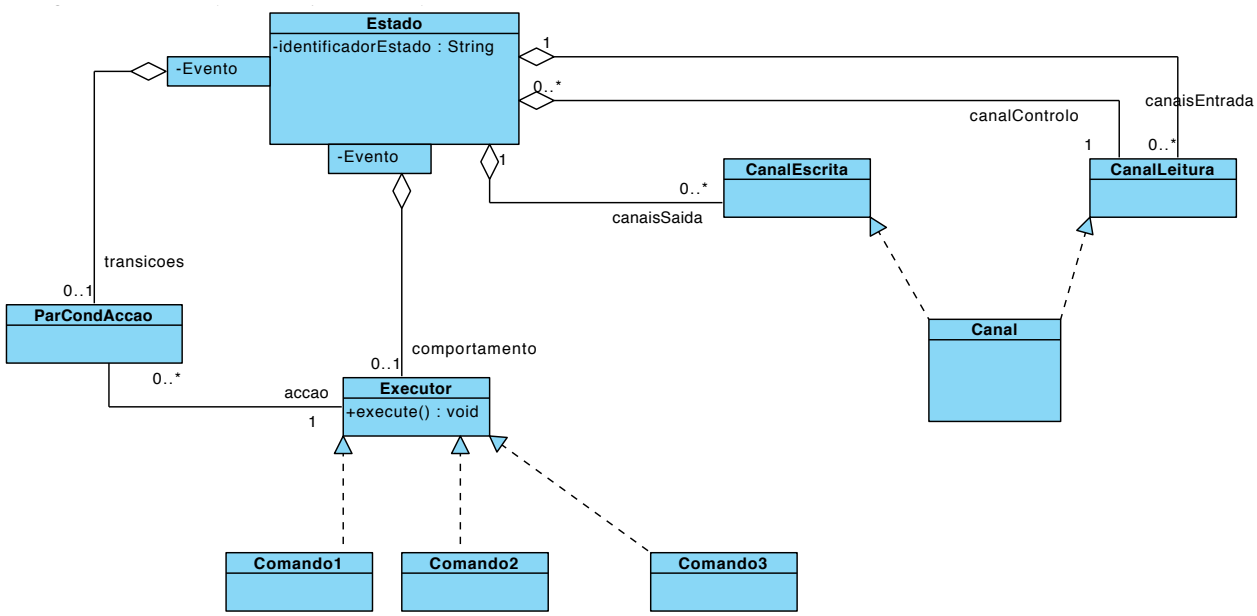


Figura 5.7: Padrão arquitectural da classe Estado.

Da imagem conclui-se que a definição de um estado é constituída pelas seguintes componentes:

- o identificador do estado;
- uma colecção de canais de entrada que definem as transições de outro estado para este;

- uma colecção de canais de saída que estão ligados aos estados destino das transições existentes;
- uma ligação de entrada para receber informação do objecto que criou e comanda a execução do diagrama de estados;
- um mapeamento de eventos, que são recebidos via canais, para objectos do tipo `ParCondAccao`. Esta variável de instância do estado tem associada uma entrada para cada transição de saída;
- um mapeamento de cada uma das actividades internas para a operação que executa a acção associada, e
- uma referência para um objecto especial que contém os atributos acedidos pelos diferentes estados. Esta classe tem um conjunto de métodos normalizados do tipo `getX` e `setX`, que possibilitam às operações do tipo `Executor` o acesso a variáveis do estado.

O construtor parametrizado da classe `Estado` tem a assinatura que se apresenta:

```
public Estado(String identificadorEstado,
              CanalLeitura[] canaisEntrada,
              CanalEscrita[] canaisSaida,
              CanalLeitura   canalControlo,
              HashMap<Evento,ParCondAccao> transicoes;
              HashMap<Evento,Executor>     comportamento;
              VariaveisEstado varEstado)
```

A animação do diagrama de estados faz-se com o recurso a instâncias de objectos do tipo `Estado`, mas é necessário que existe uma outra entidade que estabeleça a tipologia de ligações (canais) entre os estados e que também seja responsável pela criação e correcta parametrização dos estados e dos comandos (objectos do tipo `Executor`).

No caso do diagrama da Figura 5.6 o objecto de controlo é responsável por criar:

- os canais de ligação

```
Canal ligacao_A_para_B = new Canal();
Canal ligacao B_para_A = new Canal();
Canal ligacao_B_para_B = new Canal();
```

- os objecto estado

```
Estado eA = new Estado("EstadoA", ...);
Estado eb = new Estado("EstadoB", ...);
```

- o objecto que guarda os valores do estado interno
- as instâncias dos comandos e os pares condição/acção

Após a inicialização dos estados, estes ficam automaticamente activos, uma vez que o método construtor de estado invoca o método `run()`.

A definição do método `run()` é a mesma para todos os objectos estado e obedece a um algoritmo que tem os seguintes passos:

1. quando o objecto é criado fica à espera de receber informação que uma transição foi feita para ele. Enquanto tal não acontece o objecto fica à escuta, de forma não bloqueante, nos canais para determinar se existe informação para ler;
2. se o objecto receber informação nos seus canais de leitura, se a informação foi originada em `canaisEntrada`, então o objecto passa a ser o estado activo no diagrama de estados. Nessa altura o objecto deve:
   (a) verificar no mapeamento de `comportamento` se existe algum comando `Executor` para invocar;
   (b) verificar no mapeamento de `transições` se existe alguma transição automática;
   (c) se não transitou, então fica à escuta de informação do canal de controlo
3. se o objecto receber informação do canal de controlo, então foi activada uma decisão de transição e o objecto estado deve:
   (a) verificar se tem de suspender alguma actividade de `do` que esteja em execução;
   (b) executar a condição do par condição/acção que está associado à transição;
   (c) executar o comando do tipo `Executor` que está associado à actividade `exit`, se esta existir no comportamento do estado.
4. caso a transição tenha sido efectuada, o objecto fica novamente à espera de receber eventos nos seus canais de entrada.

O comportamento de um objecto que controla a animação do diagrama de estados , depois de declarar os estados e colocá-los em execução, deve interagir com o utilizador de forma a que ele decida as mudanças de estado, isto é, de modo a que seja o utilizador a determinar quando é que determinados eventos ocorrem.

Pretende-se que a animação do diagrama seja o mais próxima possível de uma utilização real num sistema interactivo (e reactivo). Quem está a animar o protótipo pode querer forçar transições não possíveis de serem habilitadas, em determinado momento, de modo a constatar o efeito que isso provoca na animação. Sempre que se pretende que uma transição seja disparada, é necessário enviar pelo canal de controlo correspondente a sinalização, através da escrita no canal com o recurso à sintaxe `canal.write(...)`. Apenas quando é recebido via sinalização de controlo, nos respectivos canais de controlo, a ordem de transição é que ela é efectuada.

Importa referir que, num determinado momento, quem está a utilizar a animação sabe sempre em que parte do diagrama de estados se encontra e o código da classe de controlo pode, se for pretendido, validar as acções que o utilizador pretende efectuar. De qualquer modo, é aconselhável que a descrição da actividade de cada estado valide se os eventos de controlo que estão a ser enviados, podem, ou não, ser considerados, o que com a definição arquitectural da classe `Estado` é simples, bastando para tal determinar se o evento que está a ser recebido pertence ao domínio do mapeamento das transições. Cada um dos estados pode ainda ser parametrizado com um modelo de apresentação próprio, de forma a comunicar ao utilizador o que se processa no estado. Este modelo de apresentação é definido por quem faz o protótipo e pode ser enviado ao objecto estado.

5.6 Reutilização do Protótipo

Uma das vantagens da utilização e construção de protótipos reside na possibilidade que a abordagem oferece de permitir antever o sistema software antes de este ser completamente construído. Através da construção de modelos que podem ser animados é possível à equipa de projecto e aos futuros utilizadores, validar os requisitos e perspectivar o sistema que ainda não foi construído. A validação por parte dos futuros utilizadores é relevante, porque constitui do ponto de vista da equipa de projecto a aceitação dos termos em que foi feita a análise, permitindo que as fases subsequentes possam ser sujeitas a menos mudanças por incapacidade na compreensão dos requisitos, ou mesmo, por ausência da recolha de alguns requisitos.

Sendo o protótipo um modelo que pode ser animado e que perspectiva uma visão do sistema previamente à existência deste, surge a questão de saber qual é o papel do protótipo na construção, em termos de desenvolvimento de código, da solução final. Uma alternativa é considerar que o modelo que se construiu e animou serviu os seus propósitos e através de técnicas bem fundadas, dará origem a uma transformação que aparece sob a forma de código final numa qualquer linguagem de programação. Neste caso, o protótipo é abandonado após ter sido utilizado e começa uma nova etapa do projecto de software que está mais directamente ligada à concepção e desenvolvimento. Se os resultados obtidos foram validados a análise de requisitos pode ser dada como encerrada, ou pelo menos, como merecedora de menor atenção a partir dessa fase.

Um outra abordagem, que deve ser considerada, é aquela que considera que o protótipo pode ser utilizado como um mecanismo de validação dos requisitos e da funcionalidade envolvida, mas o seu papel não se esgota aí [8]. O protótipo pode ser encarado como a primeira peça da construção do sistema e pode coexistir com outras peças do sistema, estas já definitivamente construídas de acordo com todos os requisitos, funcionais e tecnológicos, esperados. Esta estratégia pressupõe que o produto software que está a ser construído é numa primeira fase construído maioritariamente pelo código do protótipo e à medida que o desenvolvimento vai evoluindo retiram-se componentes do protótipo e substituem-se pelo código na sua forma final. A grande vantagem oferecida pela abordagem advém do facto de, em qualquer altura, ser possível executar o sistema e sujeitá-lo a testes funcionais. É natural que, dependendo da tecnologia envolvida, as componentes ainda pertencentes ao protótipo podem apresentar um desempenho diferente do esperado, mas a capacidade de possuir um sistema de teste funcional é uma mais valia importante.

Este desiderato é possível de ser obtido sempre que não exista uma diferença tecnológica, entre o protótipo e o sistema a ser desenvolvido, que o impossibilite. No caso dos dois artefactos serem construídos em tecnologias e modelos de programação distintos, a capacidade de utilização do protótipo para além da fase de validação de requisitos é muito reduzida, ou então, obriga à construção de suporte explícito para assegurar a correcta interligação de diferentes tecnologias e representações.

No caso desta proposta, a construção do protótipo, que permite animar os modelos construídos com o intuito de correctamente capturar os requisitos, é passível de ser incorporada na construção do sistema software desde que este seja construído em Java. Esta decisão de construção da arquitectura de suporte numa tecnologia e linguagem

[8] Esta abordagem tem vindo a ser explorada no Departamento de Informática da Universidade do Minho desde há vários anos, como patente em [Martins 95].

adequada ao desenvolvimento de soluções tem como propósito, responder a diversos requisitos de integração do processo de análise no processo global de construção do sistema software. Entre essas razões podem apontar-se:

1. o fornecimento de uma camada de suporte à prototipagem, deve assentar em conceitos próximos dos sistemas que se querem construir, de forma a promover a reutilização de elementos da arquitectura na solução final. De um ponto de vista estritamente orientado à concepção arquitectural de uma solução para um determinado problema, os componentes disponibilizados pela arquitectura de suporte fornecem um conjunto de padrões de interacção em sistemas concorrentes e distribuídos, que lhes permite fazerem parte da solução;

2. o suporte da arquitectura foi construído numa lógica de serviço a sistemas, cujo ciclo de vida é rico e dependente do estado. Os componentes da arquitectura podem ser utilizados e especializados, se necessário, para assegurarem lógicas de construção de sistemas de objectos ligados e com um fluxo de controlo orientado ao evento. Na medida em que a arquitectura corresponde à concretização de um padrão arquitectural orientado ao estado, é possível utilizar estes conceitos na modelação de sistemas cuja especificação de comportamento seja feita em função do estado em que se encontram e em que os eventos, e operações, a que reagem sejam função desse mesmo estado.

3. o facto de os objectos estarem ligados entre si por entidades que, por abstracção, se designam por canais, permite que os artefactos possam estar distribuídos lógica e fisicamente, o que introduz uma capacidade de animação dos modelos que toma em consideração outros factores, que são característica deste tipo de soluções. Na medida em que a plataforma disponibiliza componentes que permitem a invocação de serviços em objectos remotos, é possível a sua utilização na definição da lógica de negócio do sistema que está a ser desenvolvido, e

4. por último, merece especial destaque a capacidade que a transformação do comportamento, expresso sob a forma de um diagrama de estados, em código orientado à noção de estado e de evento, pode significar em termos de modelação da camada de interface do sistema a construir. A descrição do comportamento da camada interactiva pode ser feita com o recurso a diagramas de estado, onde se conseguem capturar os diferentes estados pelos quais a interface com o utilizador passa e se identificam os eventos associados e as condições em que esses mesmos eventos podem ser desencadeados [Campos 06]. Posteriormente à verificação da correcção do modelo que descreve o comportamento do caso de uso associado,

os componentes da arquitectura que suportam a construção da animação podem ser utilizados como mecanismos de controlo dos diversos elementos da interface com o utilizador (que está a ser construída). Em situações em que os vários elementos da camada interactiva não residam na mesma aplicação, ou estejam em máquinas virtuais distintas, o suporte dado pela arquitectura de suporte é especialmente válido, numa perspectiva de implementação do padrão arquitectural MVC (Model-View-Controller) [Krasner 88].

5.7 Resumo

A arquitectura proposta neste capítulo estrutura um ambiente de validação operacional dos casos de uso, na medida em que propõe uma plataforma onde os fluxos de execução podem ser validados através de prototipagem. A plataforma oferece um conjunto de mecanismos que permitem a organização dos objectos envolvidos no diálogo e a definição de como é que o diálogo entre eles se efectua.

Na definição da plataforma de prototipagem foi dada especial atenção à complexidade das interacções que a execução dos casos de uso poderia originar, e criaram-se componentes que permitem abstrair a complexidade da interacção entre objectos, garantindo que quem usa a plataforma apenas se tenha que preocupar com a declaração das entidades e com a forma como elas se relacionam.

Apresentaram-se as diversas alternativas de comunicação e explanaram-se as estratégias possíveis para aumentar a arquitectura de classes definida. Por fim, mostrou-se de que modo o diagrama de estados pode ser animado recorrendo à plataforma existente.

Capítulo 6

Avaliação da Proposta

6.1 Introdução

Este capítulo pretende ilustrar a aplicabilidade concreta do processo de modelação anteriormente proposto e apresentado. Para concretizar a aplicação do processo utilizam-se dois exemplos de sistemas que pela sua complexidade necessitam de uma sistemática abordagem às tarefas de análise. Os exemplos apresentados correspondem a sistemas marcadamente interactivos, onde a execução dos casos de uso depende da história e dos dados fornecidos pelo utilizador.

Para os exemplos utilizados, os casos de uso são detalhados e são construídos os diagramas comportamentais que especificam a sua execução. A especificação dos casos de uso é complementada com as expressões rigorosas em OCL que permitem raciocinar sobre o comportamento dos mesmos. A utilização da arquitectura de suporte à prototipagem é apresentada, como mecanismo de validação das descrições comportamentais dos casos de uso.

No fim do capítulo faz-se neste capítulo referência a outras abordagens alternativas mas com o mesmo propósito, sendo apresentadas as principais características diferenciadoras destas.

6.2 A Justificação do Processo

Apresentado que foi o processo e a motivação que norteia a sua criação, podem-se recuperar os princípios genéricos da proposta do processo Catalysis [d'Souza 99], que

são extensíveis a quem procura criar mecanismos que adicionem rigor à execução da fase de análise de um projecto software. Esses princípios são também partilhados por este trabalho e são: i) a capacidade de abstracção, ii) a capacidade de ser rigoroso e preciso no discurso e iii) a possibilidade de construir o sistema através da composição de partes e componentes.

Por capacidade de abstracção, entende-se neste contexto, a necessidade de o processo se centrar nos detalhes essenciais do sistema a construir, omitindo aspectos não relevantes. No que diz respeito à fase de análise, este conceito traduz-se na descrição concreta dos requisitos e das peças essenciais da análise, sendo que estão englobados como artefactos deste processo as regras e as entidades de negócio, os requisitos funcionais, a identificação das interacções e do comportamento das entidades.

No que concerne à introdução de rigor, o ponto a considerar tem a ver com a necessidade de encontrar notações que não provoquem ambiguidade na sua interpretação e que, por consequência, possibilitem más interpretações nas fases posteriores do processo. A possibilidade de um sistema ser construído através da composição de partes e componentes é uma característica base da orientação aos objectos e implica que a reutilização é um factor central no método de construção da solução. A reutilização de componentes, padrões e mesmo especificações, é relevante do ponto de vista da correcta gestão do processo de elaboração, construção e mesmo manutenção do sistema que se está a considerar desenvolver.

A visão que o Catalysis tem do seu papel no processo de desenvolvimento de um sistema software é partilhada por este trabalho, onde a necessidade de capturar correctamente requisitos, poder fundamentá-los de forma rigorosa e validá-los com os utilizadores é um contributo que o processo incorpora.

6.2.1 O Padrão de Separação de Abordagens

Ainda sobre esta temática, é interessante recuperar um padrão de aplicação do processo expresso no Catalysis [d'Souza 99] que aborda a separação entre os requisitos e as especificações no âmbito da análise de um sistema software. Esse padrão baseia-se num outro, apresentado no livro de Gamma et all [Gamma 94], que é designado por Bridge e que advoga que é possível manter, com vantagens, a definição separada das suas possíveis implementações.

A motivação desta preocupação com a separação de conceitos é de que faz sentido dizer que: *"os requisitos são para os utilizadores; as especificações são para os analistas e programadores"*. Os métodos, ou os processos de modelação, devem tornar esta

fronteira menos estrita, sem que contudo isso se traduza numa perda de precisão e rigor. A menor precisão e rigor pode ser consequência do facto de se estar a especificar conjuntamente com pessoas, os utilizadores, que não dominam as linguagens nem os processos de construção de modelos rigorosos.

A solução preconizada pelo padrão não implica que se caminhe no sentido de um menor rigor no processo de análise, com os documentos daí decorrentes a apresentarem maior capacidade de leitura e compreensão por parte de terceiros. Ao invés, a solução parte do pressuposto que a estratégia deve passar por manter duas vistas distintas da especificação (como o Bridge defende). Uma das vistas possui termos e conceitos que os clientes percebem e a outra é construída com representações mais formais (ou até completamente formais) que apenas os analistas de projecto são capazes de interpretar. A vista que os clientes compreendem é uma vista focada nos conceitos e regras de negócio enquanto que a vista dos analistas é mais próxima de aspectos orientados à concepção de uma solução.

Entre os dois tipos de documentação existe um continuado ciclo de interacção, com a utilização do documento formal a permitir à equipa de analistas melhorar os requisitos escritos informalmente e recolher a validação dos utilizadores. A Figura 6.1 explicita graficamente este padrão de processo que faz da separação de responsabilidades e da dialéctica entre as duas vistas uma forma de melhorar o resultado final da análise. A abordagem seguida neste trabalho partilha esta visão de que a sobre-especificação dos requisitos é determinante para se adquirir conhecimento sobre o sistema que se está a analisar, bem como para permitir validar operacionalmente a informação recolhida.

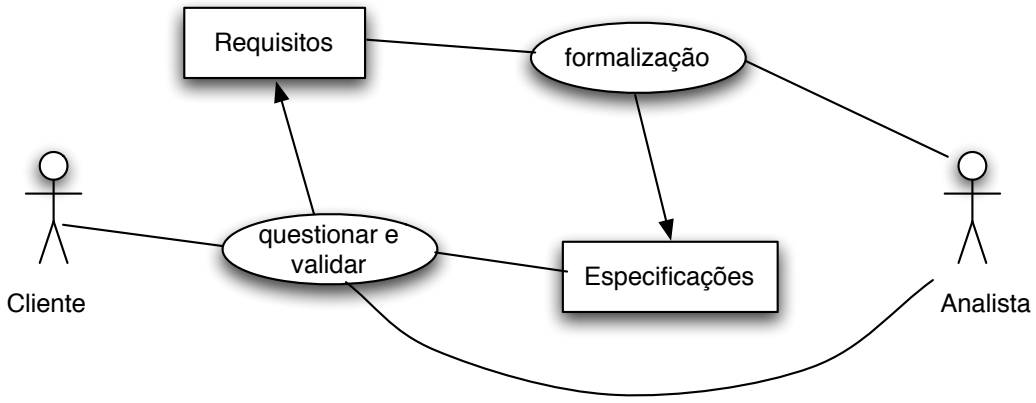


Figura 6.1: O processo de separação de vistas.

Os exemplos que se apresentam pretendem mostrar como é que este processo de construção de duas visões distintas dos requisitos de um mesmo sistema podem ser co-

ordenadas e como é que se processa a aplicação da estratégia defendida neste trabalho.

6.3 Sistema de Comércio Electrónico

Este caso de estudo é um exemplo clássico retirado de [Gomaa 00]. Corresponde à criação de um portal de comércio electrónico num ambiente tipicamente distribuído como é o caso da World Wide Web. O sistema original retratado no exemplo recorre à utilização das entidades que se espera encontrar numa aplicação destas, acrescentando-lhes uma lógica de utilização de agentes de software que fazem de intermediários entre os actores, e as suas interfaces aplicacionais. Embora neste caso esse aspecto não seja muito relevante para a lógica que está associada a uma aplicação de comércio electrónico, a utilização de mecanismos autónomos e distribuídos, como são os agentes, é algo que é facilmente modelado pela arquitectura de suporte que foi anteriormente apresentada.

No sistema de comércio electrónico existem, regra geral, clientes e fornecedores. O sistema que se está a modelar é um pouco mais elaborado, seguindo uma filosofia mais próxima de Business to Business (B2B), em que os clientes têm contratos com os fornecedores, sendo que esses contratos estão pré-estabelecidos. De um ponto de vista puramente funcional, tal significa que um fornecedor pode ter condições de fornecimento dependente do cliente em causa. Este modelo de funcionamento não é directamente visível na maioria dos portais de comércio electrónico, visto estes serem tipicamente orientados a uma lógica Business to Consumer (B2C), mas é algo que se pode encontrar em ambientes de centrais de compra. Nessa perspectiva, a utilização de agentes software que negoceiem as melhores condições contratuais é algo que se apresenta como válido e útil.

Cada cliente tem associada informação bancária, podendo ter várias contas, através das quais os fornecimentos são pagos. Os fornecedores exibem um catálogo de produtos , aceitam as encomendas dos clientes e mantém a relação dos pagamentos de cada cliente em função das compras efectuadas por estes. O cliente acede à informação dos produtos através dos catálogos e toma a sua decisão em relação aos diversos itens que pretende adquirir. Por uma questão de funcionamento do sistema, quando um cliente pretende efectuar uma compra é necessário validar se existe um contrato válido entre esse cliente e o fornecedor em causa. Cada contrato tem associada informação financeira, relativa aos fundos disponíveis para o efeito. Sempre que se pretende realizar uma compra, o sistema tem de validar se existem fundos suficientes para que essa compra seja possível. Nessas circunstâncias é criada uma ordem de entrega que é enviada para o fornecedor,

sendo que este fará toda a gestão das datas de entrega (ou de eventuais atrasos, se for o caso) e quando a encomenda está pronto o cliente é notificado e confirma quando recebe os produtos. Após o envio dos produtos comprados, procede-se à autorização do pagamento.

A Figura 6.2 apresenta de forma conceptual a organização do sistema, com os principais repositórios de dados e os agentes electrónicos existentes. Estes agentes são programas, ou componentes, que interligados através de regras de negócio, providenciam o comportamento e funcionalidade desejado.

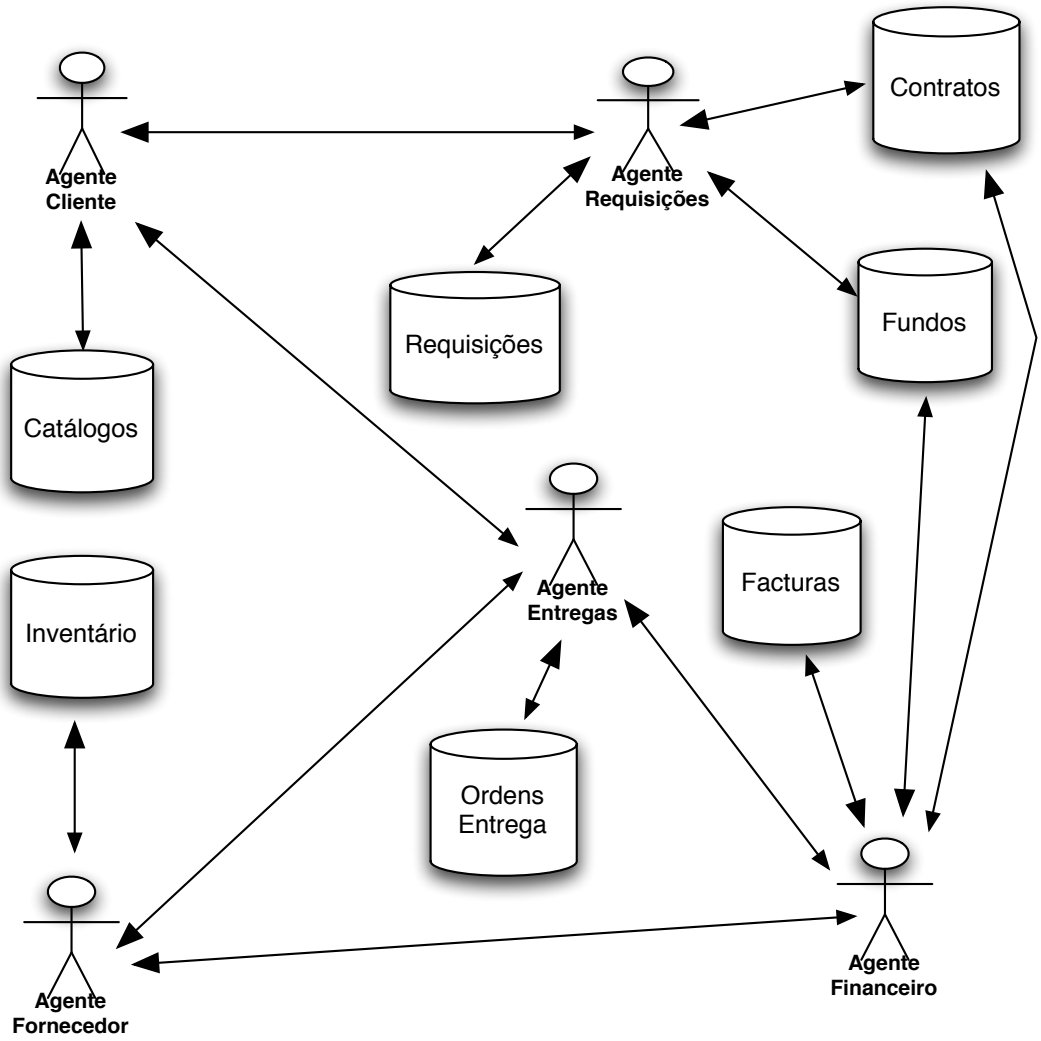


Figura 6.2: Arquitectura Conceptual do Sistema de Comércio Electrónico.

Sendo este sistema de comércio electrónico, devido à natureza dos seus interveni-

entes, um sistema corporativo, é natural que o acesso ao mesmo esteja devidamente balizado por uma gestão de credenciais rigorosa e com características de segurança mais rígidas. Acessos não autorizados são tratados de forma distinta das de um normal sistema de comércio electrónico, em que o utilizador pode estar repetidamente a tentar introduzir as suas credenciais de acesso.

6.3.1 Os Casos de Uso

A identificação dos casos de uso e a sua descrição corresponde ao primeiro passo do modelo de processo que se pretende seguir. Como é evidenciado na Figura 6.3, nesta fase do processo irão ser identificados e descritos os casos de uso. A descrição dos casos de uso faz-se com o recurso ao vocabulário normalizado que é definido pelo modelo de domínio.

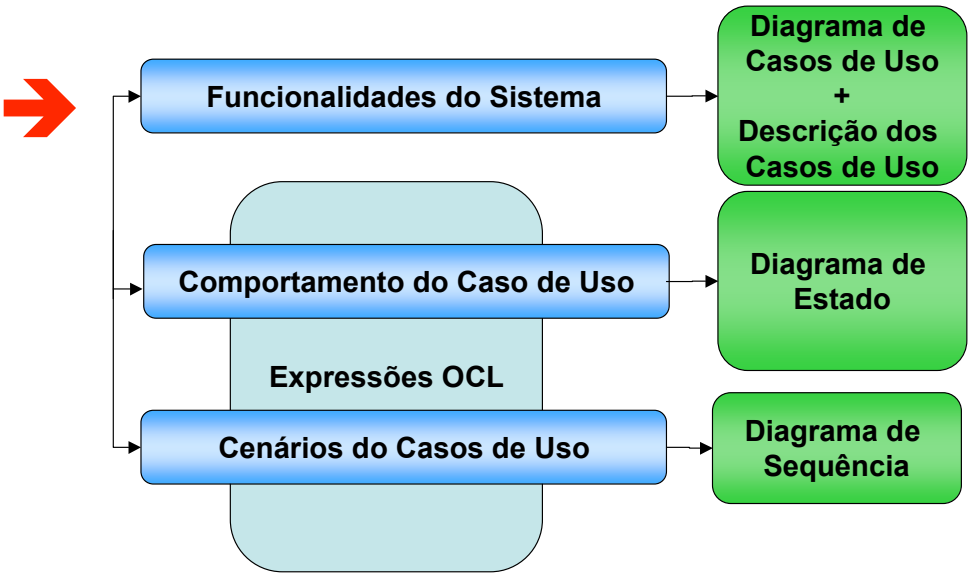


Figura 6.3: Indicador de fase do processo - casos de uso.

O modelo dos casos de uso para este sistema de comércio electrónico corporativo é apresentado na Figura 6.4. O diagrama apresenta de forma simplificada os casos de uso relevantes do sistema, na perspectiva de cliente, bem assim como os principais actores do mesmo. O actor Cliente inicia três casos de uso, "Pesquisar Catálogo", "Colocar Requisição" e "Confirmar Entrega", a que correspondem macro-funcionalidades a ser

realizadas no sistema, por parte de quem pretende adquirir produtos. Os outros actores, Fornecedor e Banco, estão associados aos casos de uso que são relevantes para a efectivação das encomendas e respectivo pagamento.

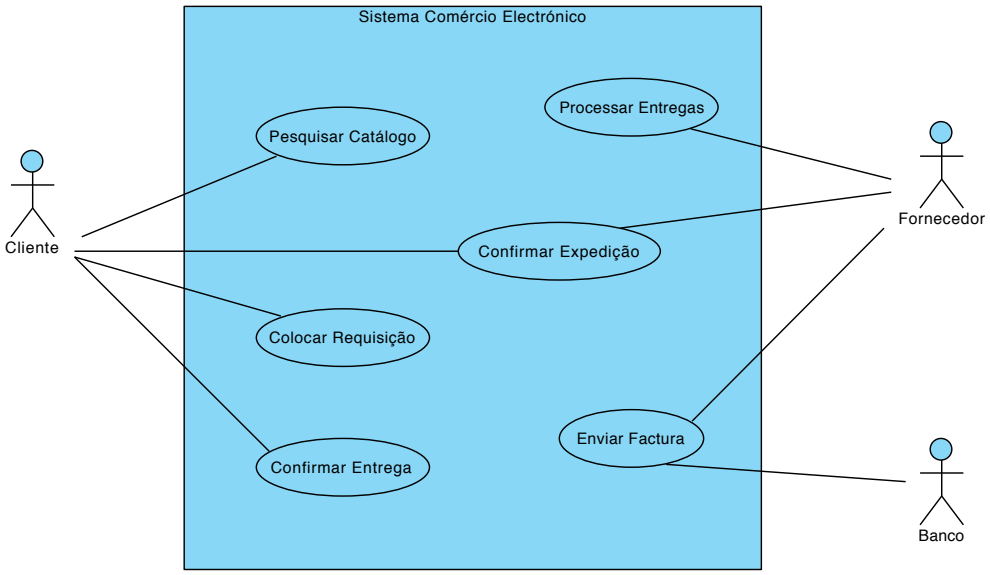


Figura 6.4: Diagrama de Casos de Uso de um Sistema de Comércio Electrónico.

No caso de uso "Pesquisar Catálogo", o cliente através de uma interface própria acede à informação dos vários itens do catálogo, fornecidos por um determinado fornecedor. O caso de uso "Colocar Requisição" é mais complexo, visto possuir uma lógica de negócio mais elaborada. Neste caso de uso, é necessário que o cliente escolha os itens que pretende adquirir e após essa escolha coloca a requisição no sistema. De seguida é necessário verificar se existe um contrato válido que permita ao cliente encomendar bens ao fornecedor, e caso exista, se o cliente tem dinheiro suficiente. Caso ambas as premissas sejam verdadeiras, a requisição é considerada aceite e o fornecedor é notificado que deve satisfazer o pedido.

Ainda no que concerne ao actor Cliente, os casos de uso que ele invoca no sistema necessitam que esteja devidamente validado o seu acesso, através do fornecimento de informação que univocamente o identifique. Por simplificação, considere-se que antes de efectuar cada um dos casos de uso associados ao actor Cliente que estão expressos na Figura 6.5, é necessário invocar o caso de uso auxiliar "Validar Acesso". Na realidade em sistemas software baseados em tecnologia Web, construídos com base num qualquer servidor aplicacional que garanta a noção de sessão e preserve a correspondente informação de funcionamento, apenas seria necessário recorrer ao caso de uso incluído

"Validar Acesso" no primeiro caso de uso que o cliente invocasse. No entanto, visto não ser possível determinar *a priori* qual é o caso de uso que primeiro efectua a validação das credenciais de acesso, é necessário que o diagrama de casos de uso preveja esse relacionamento de inclusão em todos. Da mesma forma, nas descrições dos casos de uso que a equipa de projecto tem de desenvolver, deverá ser colocada em todos, de forma explícita, a inclusão do caso de uso "Validar Acesso". Uma peça de modelação que a equipa de projecto pode utilizar, como mecanismo para registar este comportamento, é a inclusão de um diagrama de actividades onde se descreva qual é a tarefa em que um cliente se envolve quando utiliza o sistema. Esse diagrama de actividades traça as várias linhas de controlo de fluxo, bem como as actividades envolvidas, numa sessão típica de um utilizador no sistema. Se o caso de uso "Colocar Requisição" for o primeiro a ser invocado, então far-se-á aí a validação do acesso e essa informação ficará registada no sistema até que o utilizador o abandone, ou termine a sessão.

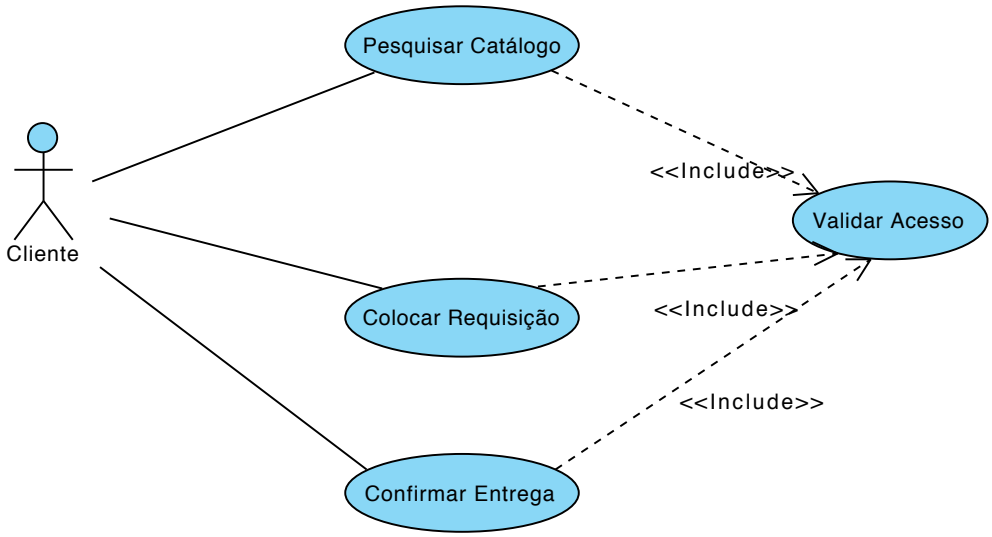


Figura 6.5: Detalhe do diagrama de casos de uso para o actor Cliente.

A descrição textual do caso de uso "Colocar Requisição" é elaborada, relacionando o conhecimento dos requisitos com os termos que o modelo de domínio define.

6.3.2 Modelo de Domínio e da Aplicação

Após se terem identificado os casos de uso principais do sistema de comércio electrónico, deve-se adquirir mais conhecimento sobre as entidades que fazem parte do domínio do

Caso de Uso	Colocar Requisição
Sumário	O cliente interage com o sistema de forma a colocar um pedido de fornecimento de determinados produtos
Actor	Cliente
Dependências	Caso de uso "Validar Acesso" deve estar incluído
Pré-condição	A form Web está em inactividade, exibindo a janela inicial de introdução de informação de credenciais
Descrição	1. O cliente fornece a informação de credenciais 2. Caso de uso "Validar Acesso" é invocado 3. O cliente escolhe no catálogo de um fornecedor os itens que pretende encomendar e solicita a criação de uma requisição 4. O sistema verifica se o cliente tem contrato com o fornecedor do catálogo seleccionado 5. O sistema identifica os contratos existentes com o fornecedor 6. O sistema pede ao sub-sistema responsável pela parte financeira a reserva dos fundos necessários para a compra poder ser efectivada 7. O sistema recebe a informação que os fundos foram cativados 8. A requisição é aprovada pelo sistema 9. O sistema cria uma ordem de entrega para a requisição 10. O sistema apresenta ao cliente o estado da requisição efectivada e volta a exibir a janela inicial
Fluxos Alternativos	2. Se as credenciais não forem validadas, o sistema informa o cliente e exibe a janela inicial 4. Se o sistema verificar que não existem contratos com o fornecedor, informa o cliente e termina o processo 6. Se a reserva de fundos não poder ser feita, por estes não serem suficientes, o processo termina e o cliente é informado
Pós-condição	A informação financeira foi actualizada com o novo valor resultante do decréscimo induzido pela requisição

Figura 6.6: Descrição do caso de uso "Colocar Requisição" num sistema de comércio electrónico.

problema e estabelecer os relacionamentos entre elas.

A Figura 6.7, apresenta o modelo de domínio do sistema de comércio electrónico, no qual se podem encontrar as principais entidades existentes e o relacionamento entre elas. O modelo ilustra os conceitos que presidem à construção de um sistema que tem um modo de funcionamento que anteriormente já se apresentou. Embora o modelo de domínio apresentado não seja a única solução possível para a estruturação dos relacionamentos entre as entidades, permite de forma bastante intuitiva representar a complexidade do problema, e retirar conhecimento sobre o modo como os diferentes conceitos se relacionam. Seria possível apresentar uma forma diferente de relacionar as entidades "Requisição" e "Item Seleccionado", de forma a tornar mais explícito que a primeira é constituída por uma colecção de itens. No entanto, a colocação da entidade "Ordem Entrega", como elemento de ligação entre ambas, permite que seja mais evidente que uma "Factura" seja directamente relacionada com a ordem, e que seja esta a guardar a informação de quais são os itens que dela fazem parte.

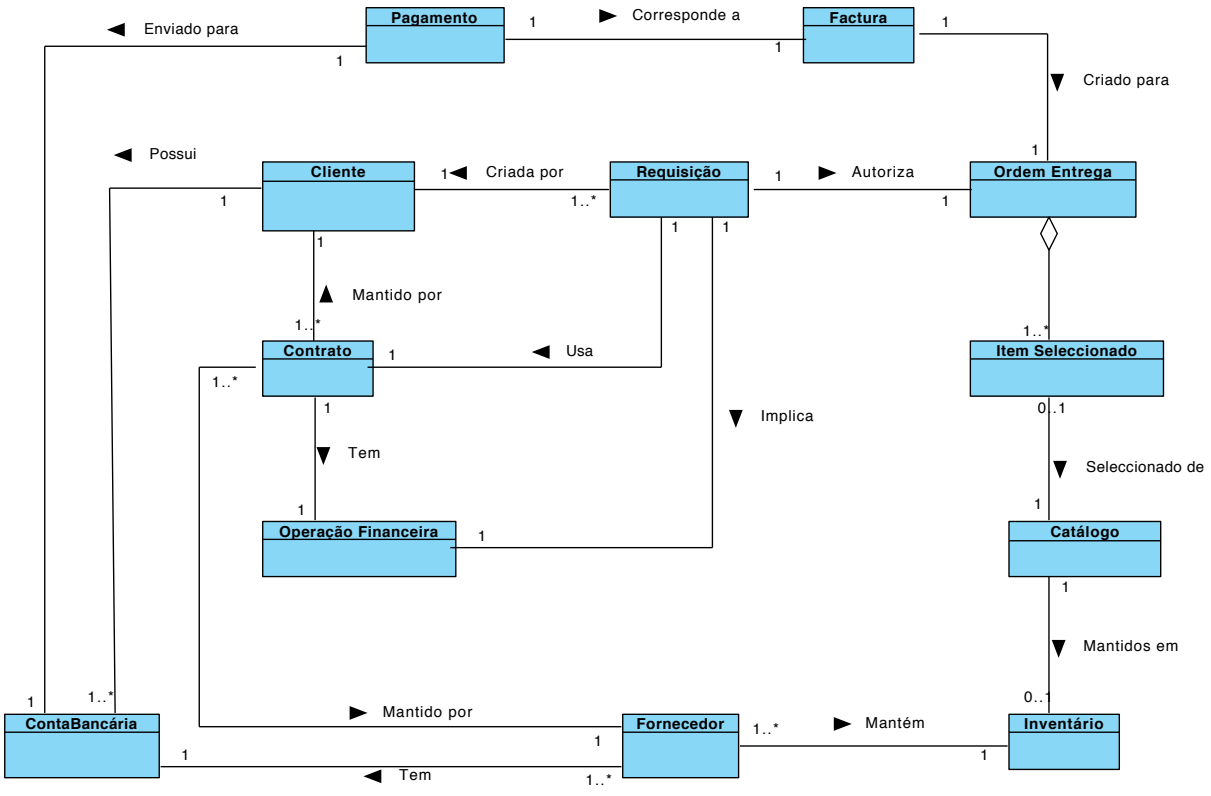


Figura 6.7: Modelo de Domínio do Sistema de Comércio Electrónico.

É possível detalhar um pouco mais a informação existente no modelo de domínio e torná-lo uma primeira versão, muito abstracta, do modelo da aplicação que vai responder à estruturação pensada para arquitecturalmente suportar o sistema. O diagrama apresentado na Figura 6.8 ilustra uma solução para a definição interna das entidades que fazem parte do sistema. Na realidade, esta pode ser considerada uma primeira versão (rudimentar) do diagrama de classes que a partir da definição inicial vai ser trabalhado de forma a modelar uma solução, que respeite os princípios básicos da programação por objectos e responda de forma adequada aos requisitos que foram identificados.

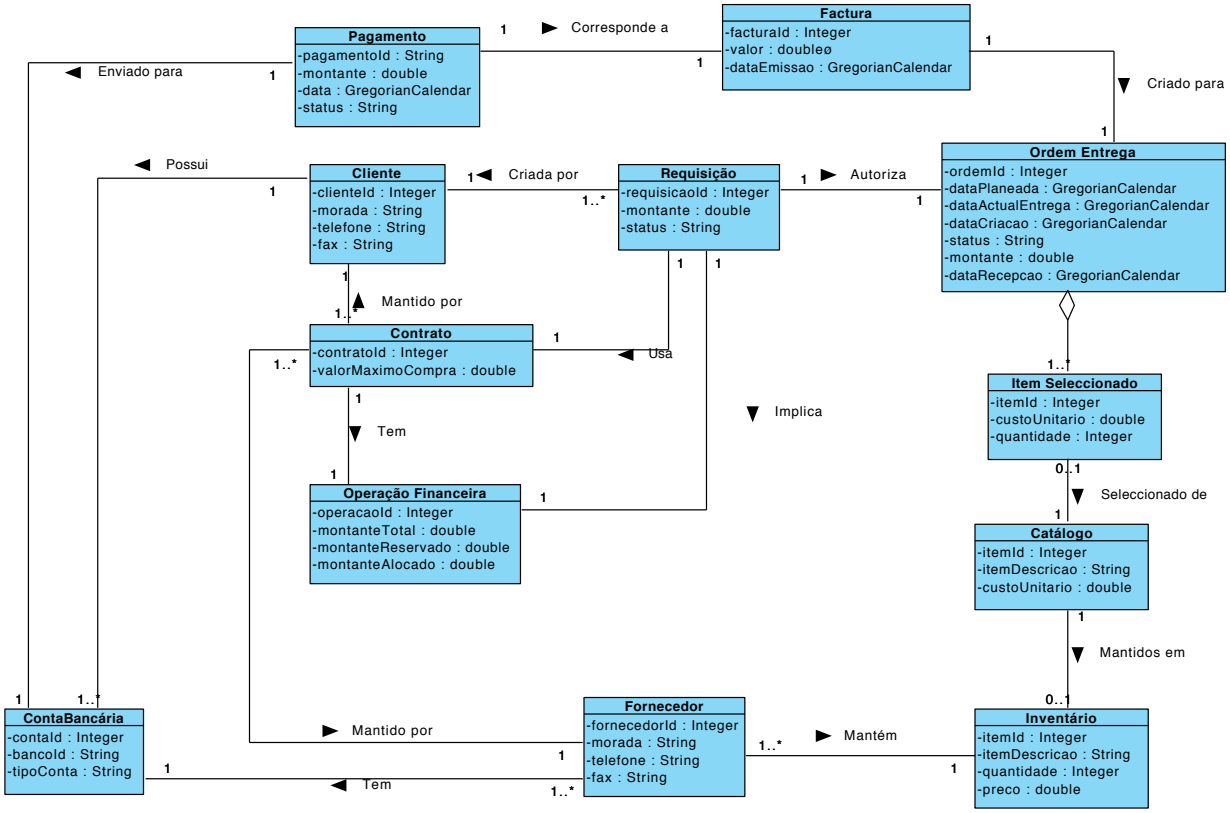


Figura 6.8: Modelo do Domínio com informação detalhada sobre as entidades.

6.3.3 O Processo de Captura de Requisitos

De forma a ilustrar a aplicação do processo, consideremos do exemplo do sistema de comércio electrónico, o caso de uso "Colocar Requisição", por parecer aquele que é

mais interessante do ponto de vista da descrição do seu comportamento.

A validação dos requisitos, quando efectuada na presença do cliente, é mais susceptível de produzir resultados quando os casos de uso em análise possuem um fluxo de execução que permita verificar se o que foi especificado corresponde ao pretendido. Nessas situações a validação faz-se por inspecção dos resultados e por análise dos mesmos se pode determinar se as pós-condições e o invariante foram respeitados. A elaboração dos contratos associados a cada caso de uso, com a especificação das asserções que devem ser verdadeiras, possibilita que se descreva quais são as restrições que devem ser respeitadas na construção do resultado. Essas restrições são na maior parte dos casos, transportadas para o diagrama de classes e permitem estabelecer condicionantes ao relacionamento entre entidades a esse nível. Outras restrições são transpostas para as operações que colectam informação e organizam as estruturas de dados que vão ser devolvidas.

A especificação, utilizando OCL, das operações envolvidas no caso de uso que efectuam a construção dos resultados é também um dos resultados deste processo. Dessa forma, a especificação das restrições garante um acréscimo de qualidade na produção do resultado, mas a verificação da correcção quer da especificação, quer da construção do resultado, implica a análise da informação. Esta é uma consequência do processo natural de teste que os clientes conduzem com a equipa de projecto, embora numa fase mais tardia do que neste trabalho é defendido. No que concerne à produção de resultados, sejam relatórios, listagens, ou outras, o papel do cliente consiste em verificar se o comportamento de geração dos mesmos foi bem descrito e desenvolvido. Essa verificação faz-se, na maioria das situações, através da construção, manual ou não, do resultado e através da comparação com o que é esperado obter. Ao acrescentar-se informação rigorosa que especifique as regras de construção dos resultados, ganha-se um aumento de qualidade substancial na produção dos mesmos, mas numa perspectiva de operacionalizar e animar a execução desse processo, são mais interessantes os casos de uso que tenham um padrão comportamental que implique a mudança de estado do sistema e a verificação, em cada momento, do estado em que o caso de uso se encontra.

Casos de uso deste tipo, reproduzem comportamentos que se encontram na camada interactiva das aplicações, bem assim como em sistemas com uma lógica de negócio partilhada entre vários componentes e que em momentos específicos se envolvem em diálogo. Nem sempre este tipo de comportamento implica que o sistema em causa seja distribuído, podendo ser, por exemplo, um sistema software embebido num servidor aplicacional, em que a lógica de construção da aplicação seja feita através da composição de componentes.

A especificação do comportamento é feita em função dos estados pelos quais o caso de uso passa, que deverão estar especificados nas expressões OCL através da expressão `oclInState(s)` e através da satisfação das pré-condições que determinam as transições de estado.

Considere-se novamente a descrição do caso de uso "Colocar Requisição" feita na Figura 6.6 descreve as acções a tomar por parte do sistema e do utilizador para a correcta execução do mesmo. Note-se que este caso de uso utiliza (inclui) um outro que efectua a validação das credenciais que estão associadas à conta. Por uma questão de simplicidade consideremos que é apenas uma chave, a *password* a ser solicitada. Se as credenciais fossem compostas por mais itens, seria necessário ter em atenção as diferentes combinações que originassem situações de erro. De forma a simplificarmos o caso de uso, considere-se que existe uma informação (a password), e que as credenciais necessárias para validar o acesso são que esta seja correctamente fornecida.

A descrição do caso de uso tal como é apresentada na Figura 6.9 incorpora já uma boa dose de rigor na forma como está estruturada, o que permite que seja comunicada à equipa de projecto muita informação importante. Note-se que mesmo na Figura 6.5 os casos de uso que estão presentes permitem admitir que existe um processo autónomo de validação das credenciais que será utilizado por outros casos de uso, como descrito no template acima utilizado como uma invocação resultante da existência de $<< include >>$.

Considere-se no entanto a diferença substancial que existe em termos da qualidade da informação, da elicitação do caso de uso visto apenas pelo diagrama da Figura 6.5 e da informação que pode estar implícita e que pertence ao domínio do problema.

Informação respeitante a:

- se as credenciais forem erradas n vezes a tarefa é terminada (mecanismo usual de sistemas online deste tipo)

- se não existirem contratos estabelecidos com os fornecedores

- se não existirem fundos suficientes para efectuar a requisição

não é evidente, se apenas considerarmos o diagrama de casos de uso e o modelo de domínio e é apenas possível de ser fornecida pelos utilizadores. O resultado do caso de uso, definido pela pós-condição que deve ser garantida, corresponde em caso de erro (credenciais erradas, inexistência de contratos, falta de fundos) às situações anteriormente apresentadas.

Caso de Uso	Colocar Requisição
Sumário	O cliente interage com o sistema de forma a colocar um pedido de fornecimento de determinados produtos
Actor	Cliente
Dependências	Caso de uso "Validar Acesso" deve estar incluído
Pré-condição	A form Web está em inactividade, exibindo a janela inicial de introdução de informação de credenciais
Descrição	1. O cliente fornece a informação de credenciais 2. Caso de uso "Validar Acesso" é invocado 3. O cliente escolhe no catálogo de um fornecedor os itens que pretende encomendar e solicita a criação de uma requisição 4. O sistema verifica se o cliente tem contrato com o fornecedor do catálogo seleccionado 5. O sistema identifica os contratos existentes com o fornecedor 6. O sistema pede ao sub-sistema responsável pela parte financeira a reserva dos fundos necessários para a compra poder ser efectivada 7. O sistema recebe a informação que os fundos foram cativados 8. A requisição é aprovada pelo sistema 9. O sistema cria uma ordem de entrega para a requisição 10. O sistema apresenta ao cliente o estado da requisição efectivada e volta a exibir a janela inicial
Fluxos Alternativos	2. Se as credenciais não forem validadas, o sistema informa o cliente e exibe a janela inicial 4. Se o sistema verificar que não existem contratos com o fornecedor, informa o cliente e termina o processo 6. Se a reserva de fundos não poder ser feita, por estes não serem suficientes, o processo termina e o cliente é informado
Pós-condição	A informação financeira foi actualizada com o novo valor resultante do decréscimo induzido pela requisição

Figura 6.9: Descrição do caso de uso "Colocar Requisição" num sistema de comércio electrónico.

Em caso de sucesso na introdução da informação de credenciais, existência de contrato com os fornecedores e fundos disponíveis, a pós-condição deve ser acrescentada da seguinte descrição:

- se as credenciais forem correctas, os contratos com o fornecedores existirem e os fundos forem suficientes, o montante necessário é cativado e é criada uma ordem de entrega.

Estas três condições constituem os elementos de validação da pós-condição do caso de uso. A tentativa de formalizar este conhecimento, mesmo que não seja usada para mais nada, garante pelo menos que o levantamento de informação é efectuado e beneficia o processo, permitindo identificar os diferentes cenários.

Assuma-se que o caso de uso "Validar Acesso" faz parte do comportamento esperado do caso de uso "Colocar Requisição". É possível, seguindo o processo proposto iniciar as tarefas de análise e recolher a informação necessária para a fase de concepção. A partir de um caso de uso são criados os diagramas de actividade[1], estado e de sequência que o modelam. Estes diagramas permitem além de especificar o comportamento do caso de uso, adquirir informação sobre as entidades necessárias a nível de conceptualização. Obtém-se, ao mesmo tempo, a capacidade de descrição dos fluxos de controlo que originam os diferentes cenários.

Após se terem identificado os casos de uso, ter sido construído o diagrama de casos de uso e ter sido registado para cada caso de uso a sua narrativa textual, o próximo passo do processo é a especificação do comportamento do caso de uso, como é perceptível no indicador de processo, apresentado na Figura 6.10.

Para a descrição do fluxo do comportamento que o caso de uso encerra, utilizamos um diagrama de estados, de modo a identificar os diferentes estados que o caso de uso atravessa e que o utilizador poderá depois validar na fase de prototipagem. O diagrama de estados possibilita também que se levantem as operações e as condições que determinam a transição entre estados, permitindo identificar os estímulos internos e externos que podem afectar a execução do caso de uso. A Figura 6.11 apresenta o diagrama de estados associado à execução do caso de uso "Colocar Requisição".

Este diagrama tem todos os fluxos de controlo que se podem encontrar no caso de uso, e permite identificar:

1. os eventos que desencadeiam a invocação das operações, e

[1]Embora estas possuam um carácter pouco rigorosos.

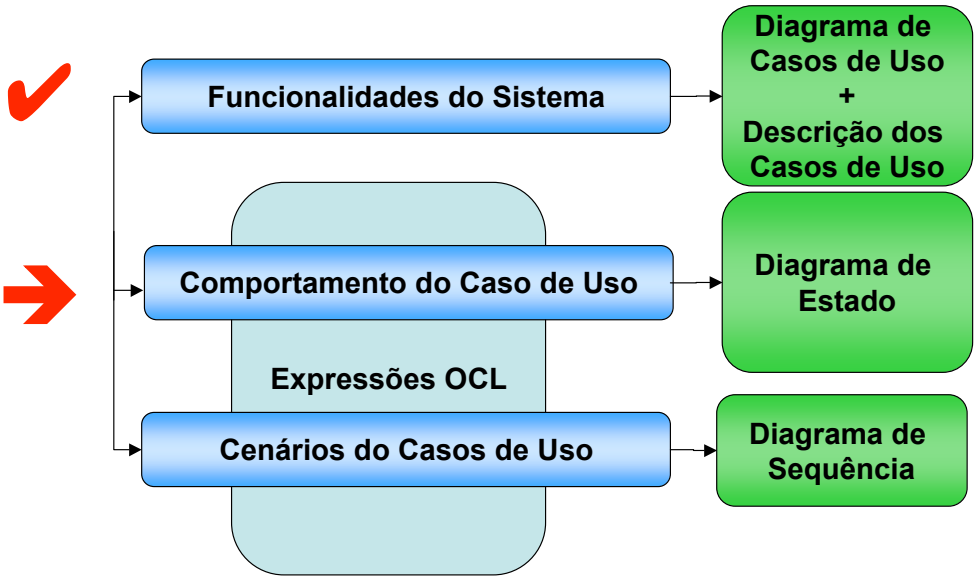


Figura 6.10: Indicador de fase do processo - modelação de comportamento.

2. as condições que determinam as transições que vão ser escolhidas dependendo dos valores obtidos.

O diagrama apresenta uma primeira versão da especificação do comportamento do caso de uso com o recurso ao formalismo das máquinas de estado. Nesta primeira versão da especificação do comportamento foram abstraídas algumas condições que devem ser melhor especificadas, como sejam as relativas à validação das credenciais de acesso, bem como é omissa a informação relativa à existência, ou não, de contratos entre o cliente e o fornecedor de quem ele pretende encomendar.

No que concerne à informação relativa aos contratos, que é feita anteriormente à requisição ser validada em termos financeiros, o caso de uso deve verificar se esses contratos existem. Como se considera que a arquitectura lógica do sistema de comércio electrónico é como se apresentou anteriormente (ver Figura 6.2), a informação que confirma se os contratos existem ou não é fornecida por uma entidade (um componente do sistema), que por simplificação considerámos que possui um método que devolve um valor booleano. A Figura 6.12, apresenta uma nova versão da especificação do comportamento na forma de um diagrama de estados, acrescentado os estados e as transições que permitem especificar a existência, ou não, de contratos estabelecidos entre o utilizador e o sistema.

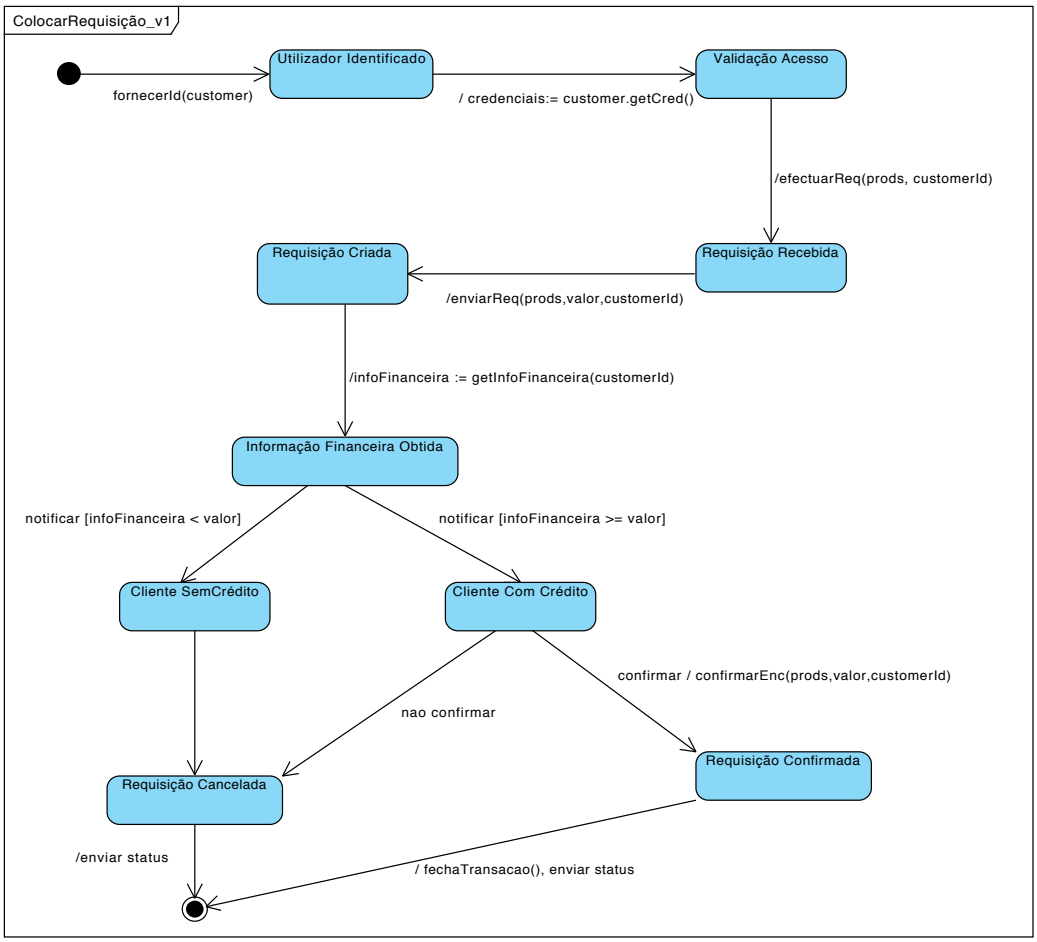


Figura 6.11: Diagrama de estado para "Colocar Requisição".

De forma a gerir a complexidade dos diagramas anteriores, ainda não se abordou em detalhe o comportamento que está associado à validação das condições de acesso de um utilizador. Não é ainda perceptível o que se passa no estado Validação Acesso e o que essa actividade implica na definição do comportamento visível do caso de uso em análise. Considere-se que esse é até um aspecto que constitui uma possível parametrização do sistema, que pode detalhar de forma diversa o que acontece durante o processo de validação dos elementos de autenticação. Assuma-se neste caso que se utiliza uma estratégia semelhante à encontrada em sistemas similares a funcionarem na Web, em que o utilizador apenas pode errar duas vezes a inserção das credenciais. Se errar três vezes essa credencial, o processo termina, com informação a ser fornecida ao utilizador (existindo por vezes um outro processo que desencadeia a criação de novas credenciais).

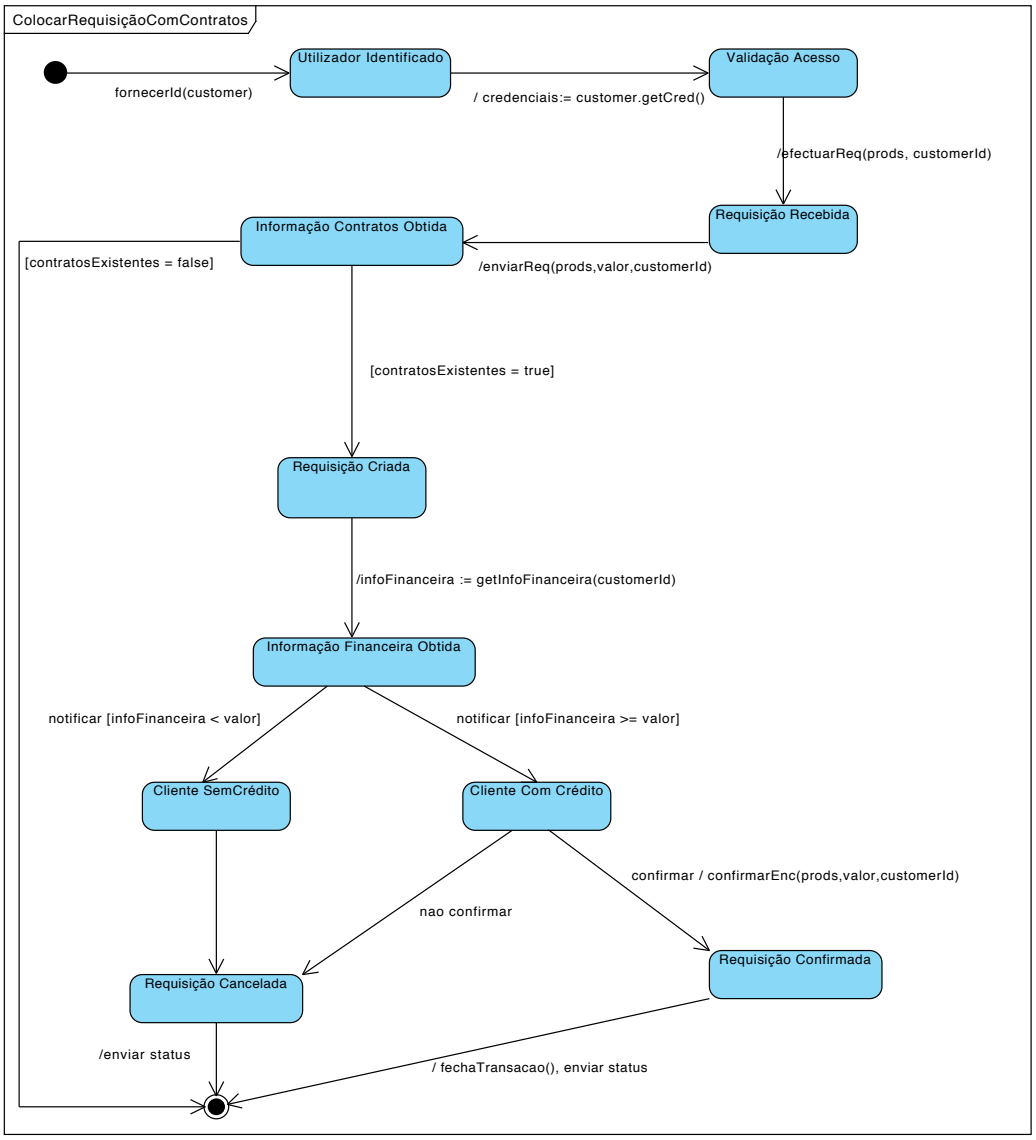


Figura 6.12: Diagrama de estado para "Colocar Requisição" enriquecido com a verificação da existência de contratos.

Por uma questão de estruturação da abordagem, consideremos o processo de validação das credenciais detalhado num diagrama de estados efectuado em separado dos anteriores. A máquina de estados que especifica o comportamento encontra-se descrita na Figura 6.13 e nela é evidente que a especificação de comportamento implica que a credencial só pode ser errada duas vezes, antes de ser desencadeado automaticamente um processo de recuperação assistida da mesma. Os métodos `fornecerId` e `getCred`, são métodos que são identificados nesta fase da especificação do comportamento e que serão posteriormente descritos e apresentados nos diagramas estruturais. O método `fornecerId` é o responsável por passar para o sistema a identificação do utilizador, enquanto que o método `getCred` é a operação responsável por adquirir a credencial que o utilizador está a introduzir. Essa credencial fica associada à variável `userCred`. Quando no diagrama se refere `userCred_2` e `userCred_3`, estas variáveis são os valores obtidos para as credenciais introduzidas pelo utilizador à segunda e terceira tentativas.

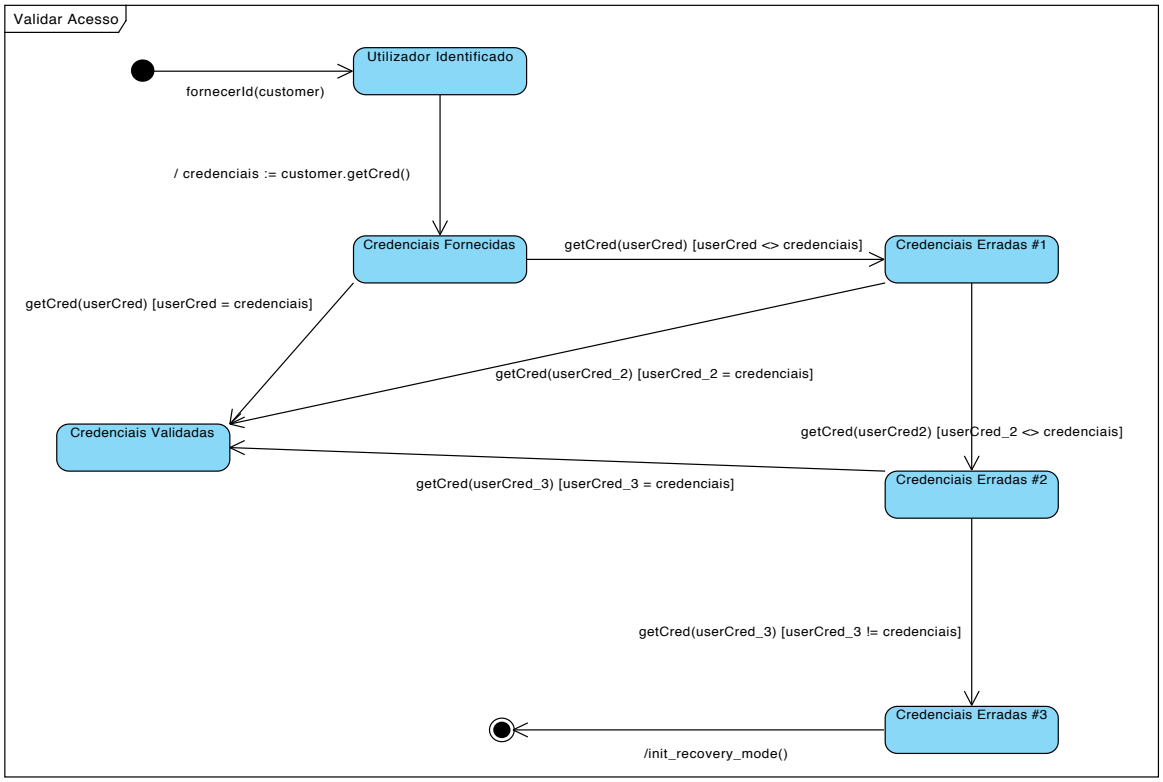


Figura 6.13: Diagrama de estado para "Validar Acesso".

O diagrama de estados que especifica o comportamento do caso de uso que faz a validação do acesso, isto é, que verifica a veracidade das credenciais para acesso ao

sistema, é suficientemente estanque para poder ser reutilizado em diversos casos de uso do mesmo sistema. Tendo em conta que o sistema de comércio electrónico é um sistema que funciona sobre protocolos e tecnologias aplicacionais Web, é possível reutilizar esta especificação de comportamento sempre que seja necessário validar as credenciais de acesso. Um comportamento usual deste tipo de sistemas, consiste em deixar os utilizadores executar as funcionalidades oferecidas pelos casos de uso e quando é necessário efectuar operações que necessitem autenticação e validação de credenciais, invocar o caso de uso, sem que se perca a informação da tarefa que estava em execução. Um exemplo comum é encontrado nos sistemas de comércio electrónico, nos quais é possível a um utilizador compor o seu carrinho de compras, mesmo sem estar autenticado no sistema, sendo que quando pretende efectuar o pagamento, a informação que está em sessão do servidor aplicacional não se perde enquanto se procede à validação do acesso. A nível de especificação esse comportamento é descrito através da inclusão da máquina de estados que descreve o caso de uso "Validar Acesso". Como mecanismo de modularidade pode-se utilizar a decomposição hierárquica existente nos diagramas de estado, de forma a utilizar a definição de caso de uso sempre que necessário.

Especificação de Comportamento de "Colocar Requisição"

Tendo em conta os vários passos de especificação de comportamento que foram anteriormente descritos para o caso de uso "Colocar Requisição", é possível detalhar num único diagrama de estados todo a especificação de comportamento do mesmo. Consegue-se assim uma percepção da complexidade do caso de uso e também se obtém a configuração das ligações entre os diferentes objectos que vão representar os estados na utilização da camada de prototipagem.

Em relação ao diagrama de estados que representa o caso de uso incluído "Validar Acesso", este poderia ser representado como um sub-diagrama de estados, por uma questão de não tornar o diagrama excessivamente complexo. No entanto, optou-se por não ter estruturação hierárquica na representação do diagrama, para ser mais fácil visualizar os diferentes caminhos que se podem traçar entre o estado inicial e o estado final do caso de uso.

O diagrama de estados completo para o caso de uso "Colocar Requisição" está expresso na Figura 6.14 e apresenta todas as possibilidades de controlo de fluxo, tendo em conta as variantes algorítmicas que foram consideradas.

A animação do diagrama de estados, permite aos utilizadores, em conjunto com a equipa de projecto, a validação dos vários diálogos que o comportamento descrito

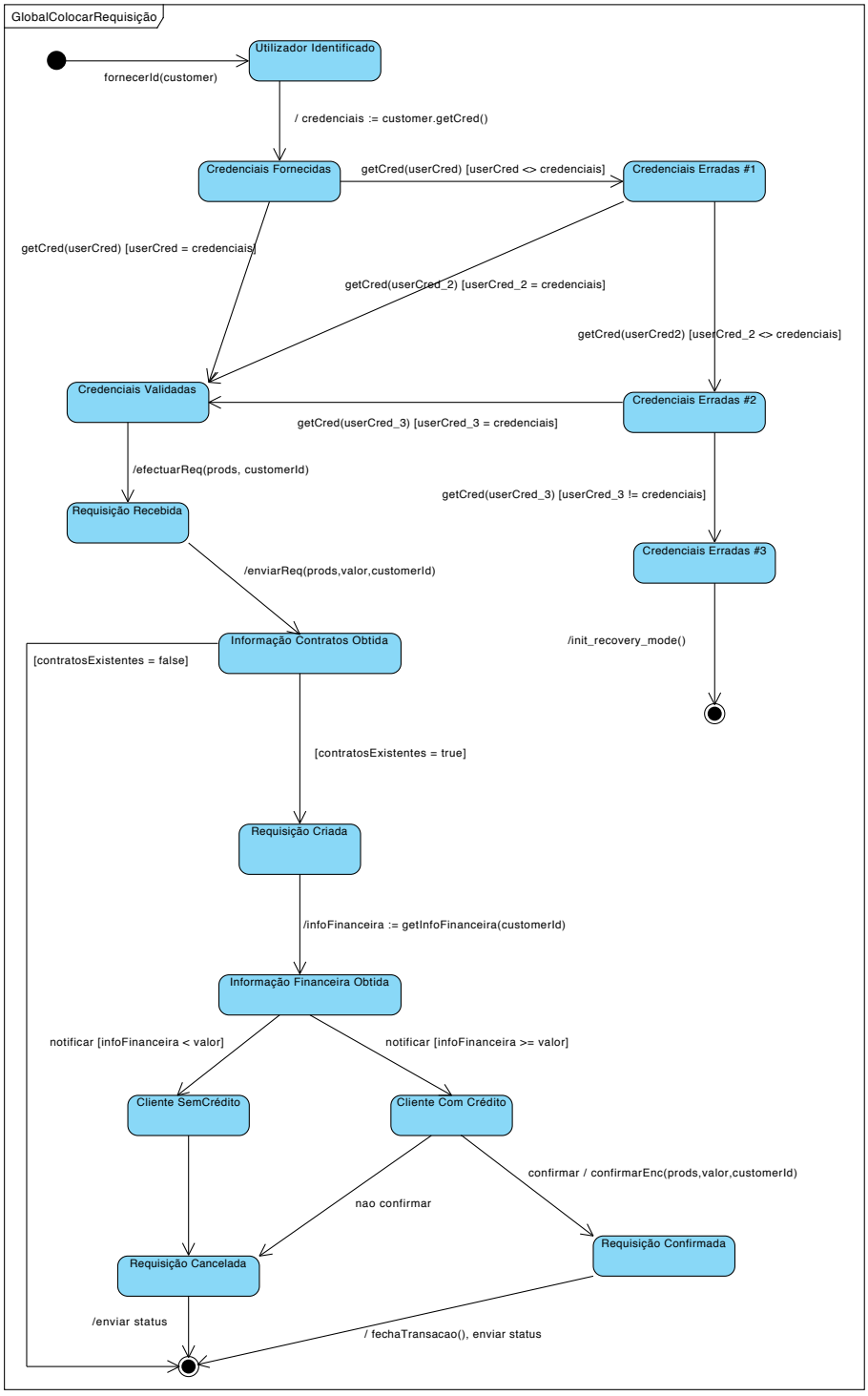


Figura 6.14: Diagrama de estado completo para "Colocar Requisição".

possibilita. Permite também verificar e validar se o percurso entre os diversos estados do diagrama é correcto e possibilita que este possa simular o envio de eventos, de forma a perceber da correcção da especificação que foi efectuada.

Após ter sido descrito o diagrama de estados que especifica o comportamento do caso de uso e onde estão explícitos todos os fios de execução, o próximo passo do processo corresponde à identificação dos cenários e à construção das interacções, expressas sob a forma de diagrama de sequência, como é proposto no indicador da Figura 6.15.

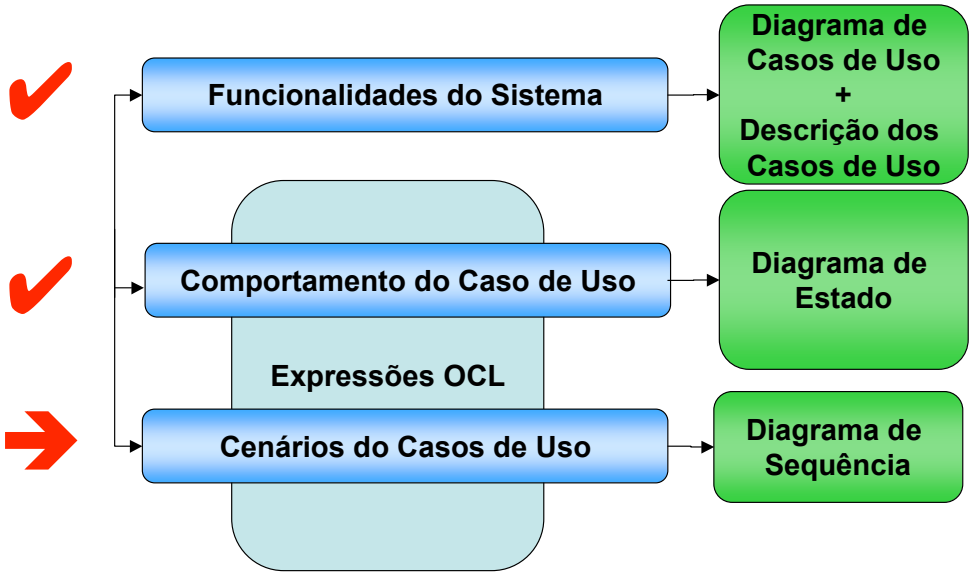


Figura 6.15: Indicador de fase do processo - descrição dos cenários.

No entanto, a passagem da análise para a construção do sistema não se faz apenas baseado na especificação de comportamento patente no diagrama de estados, mas recorre também à utilização de conceitos mais próximos das linguagens e modelos de programação. Daí que seja possível extrair do diagrama de estado os diferentes cenários que a especificação de comportamento encerra, permitindo a construção quer de outros modelos de especificação comportamental, quer a criação de cenários de teste executáveis. Esse outros modelos, que se podem extrair dos diferentes fios de execução do diagrama de estados, são os diagramas de sequência e permitem acrescentar detalhe associado à descoberta de informação, mas também podem ser utilizados como *scripts* de teste para determinados cenários.

O diagrama de sequência para o caso de uso "Colocar Requisição" explicita as

entidades e a troca de mensagens que entre elas existem. Ao descrever a interacção é possível, quando se constroem os diagramas de sequência, descobrir as classes principais que fazem parte do domínio da aplicação e que são necessárias para concretizar os conceitos e entidades expressas anteriormente no modelo de domínio. Com o decorrer do processo descobrir-se-ão outras classes que fazem parte do domínio da aplicação e que são necessárias para a resolução do problema como entendido a partir dos requisitos postos à partida e da informação dada pelos utilizadores[2].

O diagrama de sequência que descreve o cenário normal do caso de uso "Colocar Requisição" encontra-se na Figura 6.16. O diagrama apresenta as entidades principais envolvidas na interacção e faz reflectir na sua descrição o modo de funcionamento que foi apresentado. A construção do diagrama identifica as instâncias necessárias para a concretização do diálogo e se vista numa óptica não centrada exclusivamente na camada de negócio, permite obter também informação sobre as entidades (classes) que fazem parte da camada interactiva. O actor envia as mensagens a um artefacto do sistema que é a interface com o utilizador, que rege o diálogo e que comanda a execução do caso de uso. A classe InterfaceCliente assegura a gestão da comunicação com o cliente do serviço e com as entidades computacionais existentes no sistema, além de providenciar a gestão dos mecanismos que asseguram a camada de apresentação, para aquela parte da funcionalidade.

A interacção que se apresenta no diagrama começa com o cliente a identificar-se através da camada de apresentação e a fornecer as credenciais de forma a completar o processo de autenticação. Note-se que por questões de simplificação do diagrama e para apenas especificar o cenário em que todas as restrições são satisfeitas, não se apresentam as trocas de mensagem que são desencadeadas quando o cenário normal não pode ser executado.

A classe que faz a interface com o cliente, começa por instanciar um objecto da classe Cliente, para poder obter a informação que o sistema guarda sobre as credenciais e permitir validar com os dados que o actor fornece ao sistema. Após a autenticação ter sido efectuada com sucesso a classe encarregue da gestão da interface com o cliente recebe, por parte deste, a lista dos produtos que pretende encomendar. Por simplificação não se considerou o detalhe da construção da requisição, isto é, a pesquisa e colocação na requisição dos produtos que se pretendem encomendar. Essa construção é nitidamente cíclica e ao nível do diagrama de estados tem correspondência na existência de uma transição para o próprio estado, enquanto um evento que sinalize o fim deste

[2]Informação relativa ao ciclo de vida dos objectos, grau de relacionamento, etc., não está muito vezes explicitada numa primeira fase.

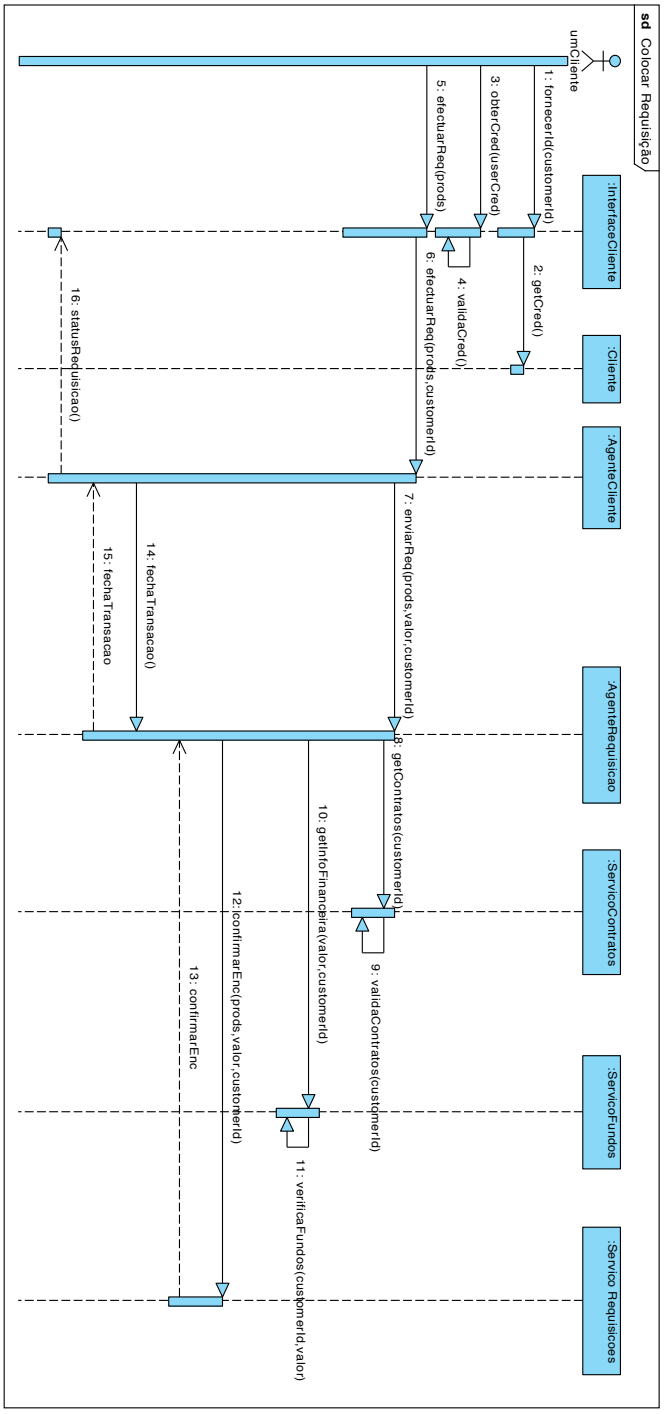


Figura 6.16: Diagrama de sequência para o caso de uso "Colocar Requisição".

ciclo não for enviado. Ao nível do diagrama de sequência seria necessário descrever uma frame de interacção do tipo loop, o que implicaria a necessidade da representação gráfica de instâncias de Item Seleccionado, conforme o modelo lógica da Figura 6.8.

Após a recepção da requisição de encomenda, o processo prossegue através do envio da requisição para os objectos que funcionam como agentes das diversas entidades. É verificado pelo objecto responsável pela validação dos contratos que estes existem e o objecto que verifica se existem fundos efectua um teste semelhante, para determinar a existência de verbas que possibilitem a prossecução da encomenda. Por uma questão de simplificação do diagrama, não se representa o que pode acontecer caso alguma destas verificações falhe. Note-se que a construção do diagrama de estados permitiu identificar o método `getInfoFinanceira`, que é utilizado para determinar a situação financeira de um cliente, bem como o método `confirmarEnc` que faz a entrega definitiva da requisição ao objecto responsável por gerir o processo subsequente de escalonamento e entrega dos produtos.

A construção do diagrama de sequência foi feita com base no pressuposto comportamento distribuído que se supõe que a aplicação terá, com vários componentes autónomos a serem responsáveis por responderem aos pedidos que lhes são solicitados. Nessa óptica os objectos do tipo, ServicoContratos, ServicoFundos e ServicoRequisicoes, são tipicamente objectos activos, ou componentes autónomos, e funcionam como elementos de interface para os seus sub-sistemas. Este aspecto é relevante, porque permite que a arquitectura de prototipagem seja utilizada para animar diálogos não síncronos e dependentes de condicionantes vários que se queiram induzir no ambiente de prototipagem.

No entanto, o diagrama de sequência apresentado como apenas modela o cenário normal, logo tem uma visão reduzida e ideal do caso de uso, apresenta alguns problemas na qualidade da informação que consegue capturar. Tendo em conta a pós-condição definida para este caso de uso, facilmente se conclui que esta não é possível de ser inferida exclusivamente a partir deste diagrama, porque:

- não é possível perceber o que se passa no processo de autenticação e em que circunstâncias é que este processo termina com sucesso

- não é evidente o que acontece se não existirem contratos entre o cliente e o fornecedor dos produtos escolhidos

- não se representa o que acontece se não existirem fundos suficientes para a colocação da requisição de compra

Apesar de a especificação dos últimos itens não ser de difícil representação no diagrama de sequência, já a componente que tem a ver com o processo de verificação das credenciais é de representação mais complexa e menos intuitiva. Apesar de ser possível colocar frames de interacção nos diagramas de sequência, este ficaria bastante detalhado, sendo que o ciclo de validação de credenciais tornaria o diagrama excessivamente complexo, sem que isso se reflectisse decisivamente na modelação (até porque o diagrama de estados já descreve essa iteração). Já no que respeita à verificação da existência de contratos ou da existência de fundos, a inclusão de frames do tipo `alt`, ou mesmo `opt`, constitui-se como a solução mais acertada.

Fios de Execução

Para o fluxo *normal* de execução a pós-condição é dada pela invocação da operação `efectuarReq`, por esta ser a última operação a ser invocada pelo cliente, embora na sequência dela outros métodos sejam invocados. Nos outros cenários, de insucesso, não é explícito no diagrama de sequência qual é a pós-condição. O diagrama de sequência apresentado não permite identificar todas as linhas de execução possíveis, visto ter sido criado para o cenário normal. No entanto, é possível criar tantos diagramas de sequência quantas as diferentes linhas de execução possíveis, de forma a analisar com mais detalhe a troca de mensagens entre os objectos.

A peça de modelação que permite raciocinar mais detalhadamente sobre os distintos fios de execução é o diagrama de estados. Pela análise do diagrama de estados da Figura 6.14, que apresenta a expansão para todos os fios de execução, é possível determinar quais são os caminhos possíveis, isto é, os cenários possíveis no caso de uso.

O diagrama apresenta vários fios de execução, a saber:

1. credenciais correctas à primeira tentativa, contratos existentes e fundos superiores ao valor da encomenda;
2. credenciais correctas à primeira tentativa, contratos existentes e fundos inferiores ao valor da encomenda;
3. credenciais correctas à primeira tentativa e contratos inexistentes;
4. credenciais incorrectas à primeira tentativa, correctas à segunda, contratos existentes e fundos superiores ao valor da encomenda;
5. credenciais incorrectas à primeira tentativa, correctas à segunda, contratos existentes e fundos inferiores ao valor da encomenda;

6. credenciais incorrectas à primeira tentativa, correctas à segunda e contratos inexistentes;

N ..., e

N+1 credenciais incorrectas à primeira, segunda e terceira tentativas.

Para os N primeiros fluxos a última operação a ser invocada é `enviarReq`, que envia a requisição para o sistema, enquanto que para o sétimo fluxo é `getCred`, que fornece ao sistema as credenciais fornecidas pelo cliente. A pós-condição do caso de uso é obtida pela conjunção de todas as condições até à última operação acrescida da pós-condição da última operação. A pós-condição de `getCred` apenas tem de se preocupar com o facto de se ter falhado a introdução das credenciais correctas o número de vezes que está determinado no processo de validação, que neste caso são três vezes.

A pós-condição de `getCred` deve determinar o estado em que o caso de uso se encontra no processo de validação das credenciais. Considerando que no diagrama de sequência apresentado anteriormente a instância de Cliente é designada por `c` e a instância de AgenteRequisicao é `aRequisicao`, a pós-condição de `obterCred` pode ser escrita como:

```
Context InterfaceCliente::obterCred(userCred:String): void
post:
   if (userCred = c.getCred()) then
     oclInState(CredenciaisValidadas)
   else if (oclInState@pre(CredenciaisFornecidas)) then
     oclInState(CredenciaisErradas#1)
   else if (oclInState@pre(CredenciaisErradas#1)) then
     oclInState(CredenciaisErradas#2)
   else
     not aRequisicao^fechaTransacao()
```

Esta pós-condição garante que se sabe sempre qual é o estado do caso de uso "Validar Acesso" em que o processo de validação se encontra. A expressão `not aRequisicao^fechaTransacao()`, significa que a mensagem `fechaTransacao` não foi enviada ao objecto em questão, porque a execução do caso de uso não chegou a esse ponto.

A pós-condição de `enviarReq` é mais complexa, visto necessitar de verificar mais condições como sejam a informação dos contratos e a disponibilidade dos fundos necessários à prossecução da encomenda. A pós-condição pode ser escrita como:

```
Context InterfaceCliente::enviarReq(produtos: Set(Produto),valor:Integer,customerId:String):void
post:
  if (sContratos.getContratos()->notEmpty()) then
     if (sFundos.getInfoFinanceira(valor,customerId) > valor) then
       sRequisicoes^confirmarEnc(produtos,valor) and
       sFundos.getInfoFinanceira(valor,customerId) =
        sFundos.getInfoFinanceira@pre(valor,customerId) - valor and
       aRequisicao^fechaTransacao()
     else
       not sRequisicoes^confirmarEnc(produtos,valor) and
       not  sFundos.getInfoFinanceira(valor,customerId) =
        sFundos.getInfoFinanceira@pre(valor,customerId) - valor and
       not aRequisicao^fechaTransacao()
   else
     not aRequisicao^fechaTransacao()
```

, em que `sContratos` é a instância de ServicoContratos, `sFundos` é a instância de ServicoFundos, `sRequisicoes` é a instância de ServicoRequisicoes e `aRequisicao` é instância de AgenteRequisicao.

No caso de não existirem contratos válidos, então nada se modifica no sistema e não chega a ser concluída a transacção, logo o respectivo método não é enviado.

É possível verificar o resultado para os diversos fluxos de execução tendo em conta os diagramas atrás apresentados e as pós-condições escritas.

Para o fluxo esperado de execução, em que as credenciais são correctas e existem contratos e fundos disponíveis, a pós-condição pode ser expressa por

$$Cond(p_1) = (userCred_1 = credencial \wedge contratos \neq \emptyset \wedge fundos \geq valor)$$

sabendo que a pós-condição da última operação é dada pelo resultado de `enviarReq`, então $final(p_1) = enviarReq(produtos, valor, customerId)$.

O que significa que simplificando, temos

```
userCred1 = credencial
and contratos->size() > 0
and sFundos.getInfoFinanceira(valor,customerId) > valor
and sRequisicoes^confirmarEnc(produtos,valor)
and sFundos.getInfoFinanceira(valor,customerId) =
    sFundos.getInfoFinanceira@pre(valor,customerId) - valor
and aRequisicao^fechaTransacao()
```

Idêntico exercício pode ser feito para os N caminhos possíveis que se identificam no diagrama de estados. Para a situação em que a credencial é correcta, existem contratos estabelecidos mas não existem fundos suficientes para concretizar a encomenda, o que se obteria seria:

```
userCred1 = credencial
and contratos->size() > 0
and sFundos.getInfoFinanceira(valor,customerId) < valor
and not sRequisicoes^confirmarEnc(produtos,valor)
and sFundos.getInfoFinanceira(valor,customerId) =
    sFundos.getInfoFinanceira@pre(valor,customerId)
and not aRequisicao^fechaTransacao()
```

De igual forma é possível descrever o que acontece quando o utilizador erra a introdução das credenciais de acesso, o que corresponde à última hipótese de fio de execução que se retira do diagrama de estados, de acordo com a enumeração feita acima.

Nessa situação temos que

$$Cond(p_{N+1}) = (userCred \neq credencial \wedge userCred2 \neq credencial \wedge userCred3 \neq credencial)$$

, o que significa que o diagrama de estado objecto ficou no estado CredenciaisErradas#3, correspondendo à expressão em OCL `oclInState@pre(CredenciaisErradas#2)`. Nessa situação o valor da pós-condição é dado por

```
not aRequisicao^fechaTransacao()
```

O raciocínio levado a cabo nesta secção permite inferir conhecimento a partir da justaposição de expressões escritas numa linguagem rigorosa. Permite também à equipa de projecto aproveitar a fase de análise para recolher dados e avançar na conceptualização do sistema.

O processo que se defende neste trabalho implica que a tarefa de análise seja levada até à construção das expressões rigorosas para a definição dos casos de uso dependentes de estado, como mecanismo de raciocínio sobre o comportamento dos mesmos. A construção das expressões rigorosas permite também à equipa de projecto validar o processo de animação do protótipo e ter uma base formal para discutir com os utilizadores os resultados obtidos.

A Figura 6.17 apresenta como proposta um arquétipo de documentação dos casos de uso, onde além da informação usualmente registada, se conjugaria também a descrição

rigorosa em OCL da formulação de comportamento e restrições existentes. Desta forma, conjugam-se na mesma peça de documentação informação de diferentes formatos e diferentes abordagens, possibilitando que se registe informação mais abrangente.

Caso de Uso	Colocar Requisição
Sumário	O cliente interage com o sistema de forma a colocar um pedido de fornecimento de determinados produtos
Actor	Cliente
Dependências	Caso de uso "Validar Acesso"deve estar incluído
Pré-condição	A form Web está em inactividade, exibindo a janela inicial de introdução de informação de credenciais
Descrição	1. O cliente fornece a informação de credenciais 2. Caso de uso "Validar Credenciais"é invocado

```
Context InterfaceCliente::getCred(userCred:String):
void post:
if (userCred = c.getCred()) then
 oclInState(CredenciaisValidadas)
else if (oclInState@pre(CredenciaisFornecidas))
then
     oclInState(CredenciaisErradas#1)
     else if (oclInState@pre(CredenciaisErradas#1))
then
            oclInState(CredenciaisErradas#2)
            else
 not aRequisicao^fechaTransacao()
...
...
```

Figura 6.17: Incorporação de OCL na descrição do caso de uso "Colocar Requisição".

A utilização da OCL é importante para a equipa de projecto perceber melhor o caso de uso e poder recorrer a ferramentas de prototipagem que permitam comunicar de forma eficaz com o utilizador, de modo a que se possa validar a captura de requisitos que foi efectuada. A documentação produzida deve ser um instrumento que o utilizador possa ler e que preferencialmente esteja escrita na linguagem que ele conhece e na qual se expressa. A descrição formal dos casos de uso é importante para a validação destes e para a informação que é fornecida em termos de descrição das operações. Esta descrição

faz-se numa forma semanticamente próxima das linguagens de programação pelo que é expectável que na fase de desenvolvimento se incluam as regras definidas a este nível.

A recolha das pré e pós-condições é relevante para se raciocinar sobre as operações em causa, enquanto que a informação relativa ao invariante de sistema é importante para a conceptualização ao nível do diagrama de classes. É possível associar restrições às associações bem como definir aspectos importantes da organização das classes (entidades), como por exemplo a cardinalidade de uma associação ou então aspectos relevantes como a escolha entre agregação e composição.

6.3.4 A Animação dos Requisitos

O processo de análise apresentado na secção anterior teve por objectivo permitir à equipa efectuar de forma mais rica e rigorosa o processo de captura de requisitos para o caso de uso. O processo detalha a descrição do caso de uso, através da descrição do diagrama de estados que rege a sua execução. Tipicamente a equipa de projecto apenas aplicará o processo aos casos de uso mais importantes do ponto de vista da construção do sistema, isto é, aqueles cujo comportamento depende da história de interacções anteriormente efectuada. Para os casos de uso que o sistema disponibiliza e que alteram o estado interno do sistema, o processo defende que a construção do diagrama de estados permite:

- a visualização dos diferentes estados que regem o comportamento interno da execução do caso de uso;
- a identificação das condições que regem a transição entre estados;
- a identificação das actividades que são efectuadas na entrada e saída de um estado e que são reflexo das pré e pós-condições associadas à invocação de operações necessárias para a execução do caso de uso, e
- a identificação das actividades que são efectuadas durante o tempo em que o controlo de fluxo se encontra em determinado estado.

Após a construção do diagrama de estados que descreve a execução do caso de uso, é possível identificar os diversos fios de execução possíveis. Cada um desses fios de execução corresponde à concretização de um cenário que é possível de ser traçado a partir do caso de uso e origina uma troca de mensagens entre os objectos que estão envolvidos no diálogo. A construção dos diversos diagramas de sequência, um para

cada cenário, leva à identificação dos objectos necessários no sistema, à identificação das classes dos quais são instâncias e ao levantamento dos métodos envolvidos.

A definição de qual é o caminho seguido em cada uma das travessias associadas à execução do diagrama de estados, depende apenas das expressões que regulam o comportamento. Essas expressões são escritas numa linguagem de restrições, a OCL, e possibilitam que se recolha e se traduza de forma rigorosa a informação informal que os clientes e utilizadores possuem. A concretização rigorosa dessa informação é utilizada no diagrama de estado como mecanismo de controlo da execução do mesmo. A definição de quais são os contratos associados ao caso de uso, logo também associados às suas especificações de comportamento, permite adicionar as restrições existentes e que fazem parte dos diagramas de estado e de sequência. As expressões escritas em OCL serão também utilizadas nos diagramas da fase de concepção, permitindo que haja uma regulação no modo como as diferentes entidades se relacionam.

A camada de prototipagem permite animar os diagramas de estado, possibilitando ao cliente a verificação da descrição que efectuou e validar operacionalmente os fios de execução existentes.

Como se referiu no capítulo em que se apresentou a camada de prototipagem, esta assenta em três princípios base:

1. a existência de entidades activas, que vão ser utilizadas para representar os estados;

2. a noção de canal como elemento de ligação direccionado entre duas entidades, e

3. a capacidade de parametrização da ligação entre duas entidades, no que respeita ao tipo de diálogo, de protocolo, que se pretende estabelecer entre elas.

No exemplo do sistema de comércio electrónico a capacidade de estabelecer diálogos entre objectos que não se baseiem numa lógica de sincronismo permite tornar a animação operacional mais realista. Na animação da execução do diagrama de estado do caso de uso "Colocar Requisição", existem transições entre estados que correspondem à invocação de serviços de agentes software que fazem parte do sistema. Dentro de uma perspectiva tecnológica em que estes agentes são concretizados em componentes autónomos que exportam serviços Web (como por exemplo, *Web Services)*), é natural que o modelo de comunicação envolvendo o envio do método e a recepção da resposta não siga necessariamente uma lógica síncrona. A animação da execução do comportamento da máquina de estados que rege o caso de uso torna-se mais realista se esta emular as situações que ocorrerão quando o sistema estiver construído. Num

sistema software com estas características é importante explorar as situações em que os diálogos não sejam apenas síncronos, mas que se possam emular situações reais, como por exemplo aquelas em que um pedido é recebido e colocado numa fila de espera e posteriormente atendido. No sistema de comércio electrónico é possível imaginar, com algum grau de certeza, que a recepção das requisições de encomenda e o seu posterior agendamento de entrega e despacho é feito de acordo com uma lógica de comportamento baseada em fila de espera[3].

A utilização da arquitectura para animar um caso de uso parte de um conjunto de regras, que devem ser seguidas pela equipa de projecto, de forma a construir o protótipo correcto. A arquitectura de suporte foi idealizada como sendo uma camada de serviço que permite tornar o mais transparente possível a construção de sistemas concorrentes, cuja expressão de comportamento pode ser descrita por uma máquina de estados.

No Capítulo 5 ao apresentar-se a arquitectura de suporte e a forma como é possível animar os diagramas de estados apresentou-se uma solução arquitectural para a descrição dos estados e indicou-se como é que se constrói a animação do diagrama. Para se criar a animação do diagrama é necessário criar alguns componentes que são a parametrização da arquitectura de suporte para o exemplo a considerar. Entre essas peças a construir, encontram-se:

- a classe que cria os estados e estabelece a topologia entre eles. Esta classe efectua a parametrização dos estados e cria os canais que os interligam, estabelecendo se são canais de leitura e de escrita;

- as concretizações das diversas acções a serem passadas para cada estado e que definem o seu comportamento interno. Não é necessário escrever o processo de execução de um determinado objecto estado, apenas tem de ser enviadas as definições das acções associadas aos eventos que o estado pode nele ver desencadeados.

- a camada de interacção com o utilizador. O objecto que define a topologia e coloca o diagrama em execução deve ser parametrizável no que respeita à definição de qual é a interface com o utilizador que deve ser construída, de forma a que o cliente possa melhor avaliar a execução do diagrama de estados que representa o comportamento do caso de uso.

[3]Provalvelmente construída sobre uma base de dados relacional.

Pré e Pós-Condições

A componente rigorosa que é escrita em OCL captura a informação que regula a execução do caso de uso, através da definição de quais são as restrições que se aplicam de forma a que (i) as condições expressas nas pré e pós-condições das operações sejam respeitadas e que (ii) as estruturas de dados alteradas ou criadas durante a execução sejam coerentes, isto é, que respeitem o invariante de estado. Na abordagem que se segue neste trabalho o primeiro contributo apresenta-se como mais relevante, na medida em que as pré e pós-condições regulam, de forma mais evidente, o disparar, ou não, das transições existentes entre estados.

Na especificação do comportamento com recurso às máquinas de estado, isto é, aos diagramas de estado, a OCL assegura a especificação das restrições associadas às condições que habilitam as transições. No entanto, tendo em conta a natureza dos diagramas de estado, a noção de pré e pós-condição pode ser especializada, de acordo com o entendimento que é definido nas Protocol State Machines [Booch 05] e que define que existem restrições que regem as condições (guardas) da transição, as pré e as pós-condições associadas ao estado. Por uma questão de clareza na abordagem, consideremos algumas regras na utilização da plataforma de prototipagem, de modo a integrar as expressões OCL que foram especificadas. Seja assim:

1. quando a pré-condição determinar a transição de estado, então deve estar expressa no diagrama de estados como guarda da transição e deve ser testada quando se recebe, via objecto de controlo, a informação que se deve transitar;

2. quando a pré-condição não estiver exclusivamente associada à transição, mas também à verificação de variáveis do sistema, então devem-se efectuar esses testes na actividade `entry` do estado para o qual se transitou;

3. a pós-condição deve ser testada previamente a desencadear a transição para um outro estado. Para isso utiliza-se a actividade `exit`,

4. quando se pretender testar a validade do invariante, a actividade `do` pode ser utilizada para esse efeito.

A transformação destas regras para a construção do protótipo é facilmente concretizada na altura de criação dos objectos estado. As definições relativas à especificação do comportamento interno de cada estado e às transições a efectuar, são fornecidas aos objectos estado e estão associadas às variáveis de instância `transicoes` e `comportamento`,

como está expresso na solução arquitectural apresentada na Figura 5.7 do capítulo anterior.

Quando for necessário para a validação de algumas condições o acesso a variáveis globais do caso de uso, os objectos estado utilizam a variável de instância `varEstado`, que disponibiliza os métodos necessários para obter e alterar esses valores.

As expressões OCL que se definem na análise comportamental do caso de uso dão origem, na utilização da arquitectura, a comandos (instâncias do tipo `Executor`). Esses comandos são escritos com base em condições que se quer executar no âmbito de determinado estado e necessitam, na maioria das vezes, de efectuar o acesso às variáveis de estado partilhadas. A escrita dessas expressões OCL, fica facilitada pela existência no objecto que gere as variáveis partilhadas de métodos que asseguram que para cada uma dessas variáveis está disponível um método de interrogação (um `getValorDeX`) e um método de alteração (um `setValorDeX`). A transformação das expressões OCL em código final fica assim bastante simplificada.

Por vezes, as expressões escritas em OCL utilizam um mecanismo próprio da linguagem para explicitamente indicarem qual é o estado em que o controlo de execução se deve encontrar. O uso da expressão `oclInState` determina qual é o estado e quando aplicada na pós-condição, obriga a que internamente cada um dos estados tenha internamente uma variável onde guarde a sua designação (cujo valor pode ser enviado pelo construtor). Dessa forma é possível efectuar a avaliação da pós-condição através da comparação desse valor.

Um outro aspecto importante na animação do diagrama de estados é aquele que incide sobre a natureza das máquinas de estado que são consideradas. Seguindo a semântica dada na UML às máquinas de estado, bem como às suas declinações em diagramas de estados, apenas após uma transição estar completamente concluída é que é possível processar um outro evento. Ou seja, enquanto o diagrama de estados está a efectuar a execução de um método associado à transição e a verificar se é possível executá-la, não é permitido que um outro evento seja processado, isto é, não é possível executar outro método. Logo não é possível ter *interleaving* de invocações, construindo diálogos da forma

$$enviarA(), enviarB(), receberRespostaA(), receberRespostaB(),$$

sendo que depois de se enviar o pedido `A` dever-se-ia receber uma resposta resultante da execução dessa operação. A plataforma de suporte, funciona de acordo com este princípio, e após ter recebido a notificação que num determinado estado deve ser disparada uma transição procede à invocação de todos os passos necessários para que

se verifique se essa transição pode ser efectuada. Após este momento o estado fica habilitado a receber mais eventos, embora caso a transição tenha sido efectuada para outro estado (situação mais comum) o envio de eventos não vai ser considerado pelo estado, por ele já não ser o estado activo. Com este padrão de funcionamento, não é possível receber um outro evento porque o controlo de programa não está disponível para o tratar, interpretar e executar, assegurando-se assim que a semântica dos diagramas de estado é respeitada. Este arquétipo de funcionamento corresponde à concretização de um ambiente de sequencialização, que implica que os eventos após serem recebidos são tratados, não existindo nenhum mecanismo que fique à escuta de potenciais eventos que estejam a ser recebidos.

Este modo de funcionamento não implica contudo, que não possam existir actividades e transições a ocorrer em simultâneo. Assuma-se que existe um estado com sub-estados concorrentes o que possibilita que se podem modelar situações em que dois eventos podem estar a ser tratados ao mesmo tempo. No entanto, este cenário implica que o mecanismo de controlo interage directamente com os vários estados que estão nas diferentes sub-máquinas de estado, logo do ponto de vista estritamente operacional o modelo de funcionamento é coerente. O suporte que a arquitectura fornece para a criação de linhas de execução em simultâneo, é fornecido pelo construtor Par que possibilita que se enviem vários eventos através dos canais (as ligações) associados a esse componente.

6.3.5 A Construção do Protótipo

Nesta fase do processo de análise existem já diversas peças de especificação que contribuem para que seja possível afirmar que a captura de requisitos levantou um conjunto de conhecimento, importante para que se possam apreender todos os detalhes do caso de uso. Antes de se operacionalizar o modelo, existem como elementos da especificação:

1. a descrição textual do caso de uso com a identificação dos passos necessários e a descrição textual das pré e pós-condições;

2. a descrição do caso de uso com o recurso aos diagramas comportamentais UML, nomeadamente ao diagrama de estado, como mecanismo de descrição de qual é o seu controlo de fluxo, enquanto está a ser executado. O diagrama de estado, permite identificar os diversos fios de execução possíveis no caso de uso e descrever as condições em que as mudanças de estado são efectuadas. Se o detalhe fornecido pelo cliente do projecto for de modo a que se possa determinar o que acontece

em cada transição, é possível ir refinando o modelo e descobrindo as operações (os métodos) que as entidades do domínio da aplicação devem disponibilizar;

3. com base no diagrama de estados que descreve o comportamento do caso de uso, descrevem-se as interacções na forma de diagramas de sequência, sendo que é possível tirar partido da estruturação das frames de interacção, de forma a criar um diagrama de sequência para cada cenário, isto é, para cada fio de execução do diagrama de estado. A construção deste diagrama implica que sejam identificadas as entidades, normalmente os objectos, que fazem parte do diálogo necessário à execução do caso de uso. Se a construção do diagrama de estado permitiu identificar alguns métodos associados às transições então o diagrama de sequência já reflecte esse conhecimento, e

4. a descrição das operações associadas à execução do caso de uso é feita numa linguagem rigorosa, OCL, permitindo identificar as condições em que é possível efectuar as operações, ou seja, em que se procede à transição de estado e discorrer sobre o que acontece ao estado após as transições. A descrição em OCL também possibilita que se detalhe a forma como os resultados são criados e, caso existam, as condições em que se criam novos objectos.

Possuindo esta documentação, que detalha de forma substantiva o caso de uso, é possível à equipa de projecto operacionalizar a validação através da prototipagem dos modelos. A animação do diagrama de estados faz-se através da construção de um programa com recurso aos componentes disponíveis na arquitectura de suporte e de acordo com as regras de utilização definidas anteriormente. Considerando o diagrama de estados da Figura 6.14, a transposição para a arquitectura de classes implica que seja criado um objecto para cada estado do diagrama.

Todos esses estados são objectos activos, logo subclasses de `Thread` (ou compatíveis com `Runnable`) e devem aceitar, via construtor, os objectos que representam as ligações entre eles e a sua definição de comportamento interno e externo. Do mesmo modo, os objectos activos que representam os diversos estados do diagrama, devem poder ter componentes que disponibilizem algum tipo de interface com o utilizador. Neste momento a arquitectura apenas tem um mecanismo de inspecção interna básico, em modo texto, mas define o princípio da hierarquia dos componentes de visualização. A criação de um ambiente visual de prototipagem, com a capacidade de criação dos objectos e interligação dos mesmos, acrescida da capacidade de inspecção, na linha de ambientes de prototipagem como o BlueJ (www.bluej.org), é tema de trabalho futuro.

Por uma questão de simplicidade, consideremos a parte do diagrama de estados relativa à validação das credenciais. O diagrama de estados deste pedaço de comportamento está expressa na Figura 6.13. Este diagrama dá origem às seguintes declarações:

```
Estado utilIdent = new Estado("UtilizadorIdentificado", ...);
Estado credForn = new Estado("CredenciaisFornecidas", ...);
Estado credVal = new Estado("CredenciaisValidadas", ...);
Estado credErradas_1 = new Estado("CredenciaisErradas1", ...);
Estado credErradas_2 = new Estado("CredenciaisErradas2", ...);
Estado credErradas_3 = new Estado("CredenciaisErradas3", ...);
```

No que concerne às ligações entre os objectos estado, uma vez que não existe nenhum requisito especial que impeça que se utilize o comportamento síncrono, os canais a serem criados são:

- `utilIdent_para_CredForn` de `UtilizadorIdentificado` para `CredenciaisFornecidas`;
- `credForn_para_CredVal` de `CredenciaisFornecidas` para `CredenciaisValidadas`;
- `credForn_para_CredErradas1` de `CredenciaisFornecidas` para `CredenciaisErradas1`;
- `credErradas1_para_CredVal` de `CredenciaisErradas1` para `CredenciaisValidadas`;
- `credErradas1_para_CredErradas2` de `CredenciaisErradas1` para `CredenciaisErradas2`;
- `credErradas2_para_CredVal` de `CredenciaisErradas2` para `CredenciaisValidadas`;
- `credErradas2_para_CredErradas3` de `CredenciaisErradas2` para `CredenciaisErradas3`;

Falta acrescentar às ligações anteriores, a ligação com origem no estado inicial e destino no estado `UtilizadorIdentificado` e a ligação que sai de `CredenciaisValidadas` para o estado seguinte no diagrama. Os estados além das ligações identificadas recebem também uma ligação de controlo do objecto que inicializa e anima a execução.

A expressão OCL que controla a execução deste pedaço do diagrama de estados apenas efectua o teste do valor da credencial que é introduzida pelo utilizador, comparando-a com valor que está associado no sistema. Em função do resultado do teste e do estado em que se encontra o controlo de execução, é decidido qual é a transição a desencadear.

Sendo a expressão OCL correspondente, definida como sendo:

footnotesize

```
Context InterfaceCliente::obterCred(userCred:String): void
post:
   if (userCred = c.getCred()) then
```

```
    oclInState(CredenciaisValidadas)
  else if (oclInState@pre(CredenciaisFornecidas)) then
    oclInState(CredenciaisErradas#1)
  else if (oclInState@pre(CredenciaisErradas#1)) then
    oclInState(CredenciaisErradas#2)
  else
    not aRequisicao^fechaTransacao()
```

o comportamento dos objectos estado relativos à validação de acesso, deve ser transformado em código de modo a respeitar a expressão. O objecto do tipo `ParCondAccao`, que determina se a transição deve, ou não, ser habilitada, tem uma expressão de condição que corresponde a obter o valor correcto da credencial e compará-lo com o que foi introduzido pelo utilizador. A parte do teste da condição é de escrita simples e obedece à seguinte expressão:

```
public boolean testaCondicao() {
  return (this.varEstado.getCred()).equals(userCred);
}
```

A transição é efectuada após o resultado deste teste e em função dele executa-se a transição para o estado correspondente. Uma vez que não existia na definição das expressões OCL associadas à validação das credenciais, mais informação que a necessária para a efectivação das transições de estado, não é preciso acrescentar mais comportamento. Note-se que uma vez que os estados que são utilizados para mapear a execução da validação de acesso, são muito semelhantes, a sua expressão de comportamento pode ser a mesmo, necessitando apenas de verificar na expressão OCL acima apresentada qual é o estado actual para determinar para onde se transita.

Apresentam-se de seguida alguns momentos de utilização da arquitectura com o intuito de animar a execução do caso de uso. Uma vez que o caso de uso "Colocar Requisição" tem uma expressão de comportamento razoavelmente complexa, mostra-se de seguida a utilização da animação para a fase de aquisição das credenciais e validação da existência de contratos entre o cliente e os fornecedores.

A arquitectura permite que cada objecto estado possa ser instanciado com um determinado modelo de apresentação que define a forma como apresenta os resultados. No exemplo que se apresenta, o modelo de apresentação por omissão é utilizado.

A primeira situação a ser apresentada corresponde à animação da sub-máquina de estados "Validar Acesso", que encerra a lógica relativa à verificação das credenciais.

De acordo com o diagrama de estados, apresentado na Figura 6.13, a verificação das credenciais comparando a informação que o utilizador introduz com o valor da credencial que é guardado no sistema (ao qual a execução do caso de uso tem acesso). Essa validação é feita através de um número pré-determinado de tentativas, que ou resulta na validação dos dados fornecidos ou resulta no término da animação do caso de uso.

A Figura 6.18 apresenta um momento da execução em que a informação da credencial foi correctamente fornecida tendo sido activada a transição entre o estado Credenciais Fornecidas e o estado Credenciais Validadas. Os estados encarregues de efectuar novas tentativas de aquisição das credenciais, de forma a recuperar de uma situação de erro, não chegam a ser visitados porque o cenário escolhido não os contempla.

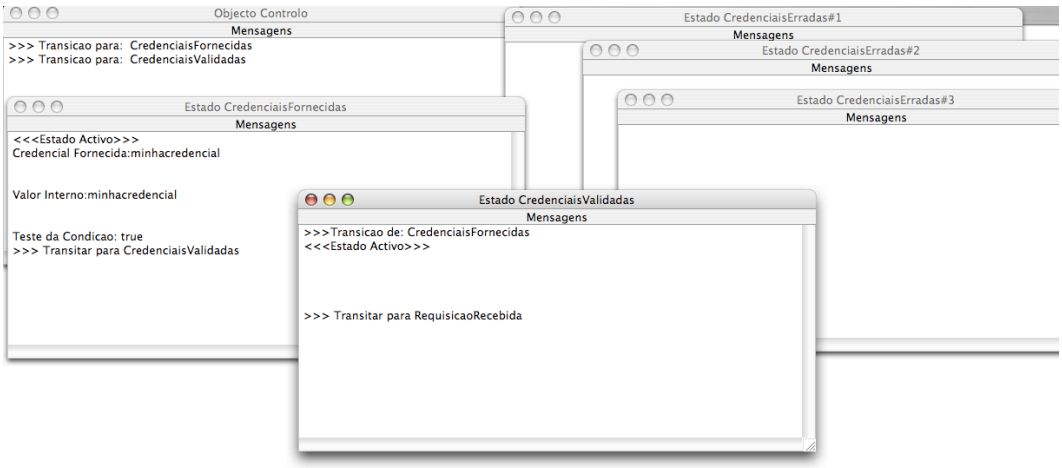


Figura 6.18: Momento de execução da validação com sucesso das credenciais.

No caso de a credencial fornecida e validada no estado Credenciais Fornecidas não ser igual à existente no sistema, então o controlo do diagrama passa para o estado Credenciais Erradas#1 em a expressão de comportamento é a mesma, como é evidente na definição OCL que foi produzida. A Figura 6.19 apresenta um exemplo da execução em que a credencial foi errada uma primeira vez e depois foi correctamente introduzida já no contexto do estado Credenciais Erradas#1.

Neste caso, em ambos os estados Credenciais Fornecidas e Credenciais Erradas#1, a expressão que determina a transição (a parte condição do objecto `ParCondAccao`) testa que `userCred = c.getCred()`, como está definido em OCL, sendo `userCred` os dados fornecidos pelo utilizador e `c.getCred()` o acesso ao valor que o sistema tem para a credencial.

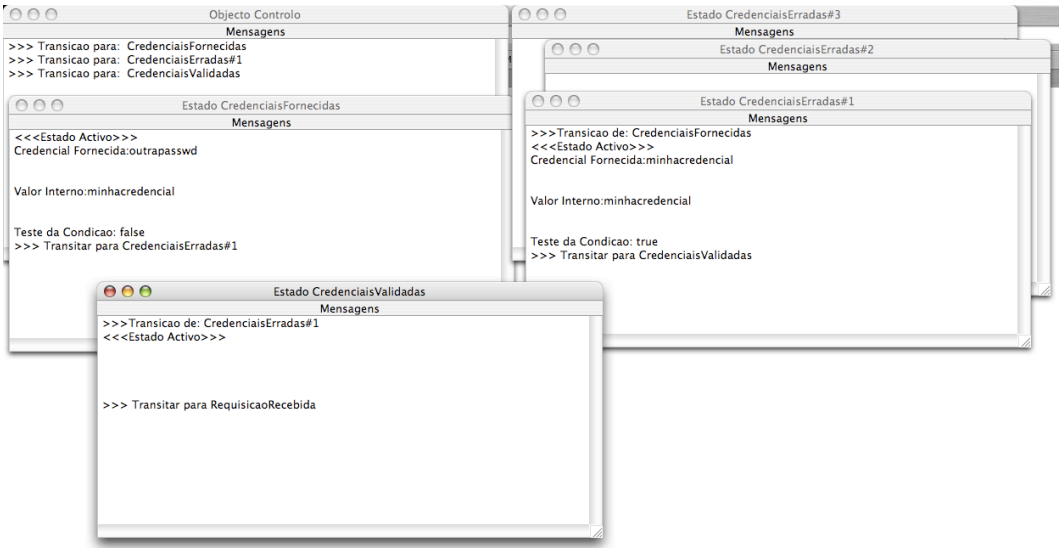


Figura 6.19: Outro momento de execução de validação das credenciais.

Uma outra situação que pode ser testada é a que determina se existem ou não contratos que permitem a concretização da compra, isto é, a colocação da requisição a um fornecedor. O estado Informação Contratos Obtida transita para outros estados em função da existência de contratos, que formam definidos como existindo se `contratos->size() > 0` fosse verdade. A execução que se apresenta na Figura 6.20, apresenta um momento de interacção em que a execução termina porque a condição não foi satisfeita.

Em função da adequação dos resultados visíveis na animação do diagrama de estados, o cliente com a equipa de projecto deve analisar os resultados obtidos. Dessa análise podem concluir que a inspecção que fizeram aos diversos cenários corre um consoante as expectativas ou então chegar à conclusão que é necessário alterar o comportamento, isto é, é necessário iterar o modelo.

6.3.6 A Iteração do modelo após validação operacional

A prototipagem do caso de uso através da operacionalização da sua expressão de comportamento permite ao cliente avaliar os resultados obtidos. Recordando o que se disse na apresentação do processo de modelação os resultados que se obtêm são de três tipos, a saber:

- resultado negativo - que surge quando uma pós-condição não é respeitada durante

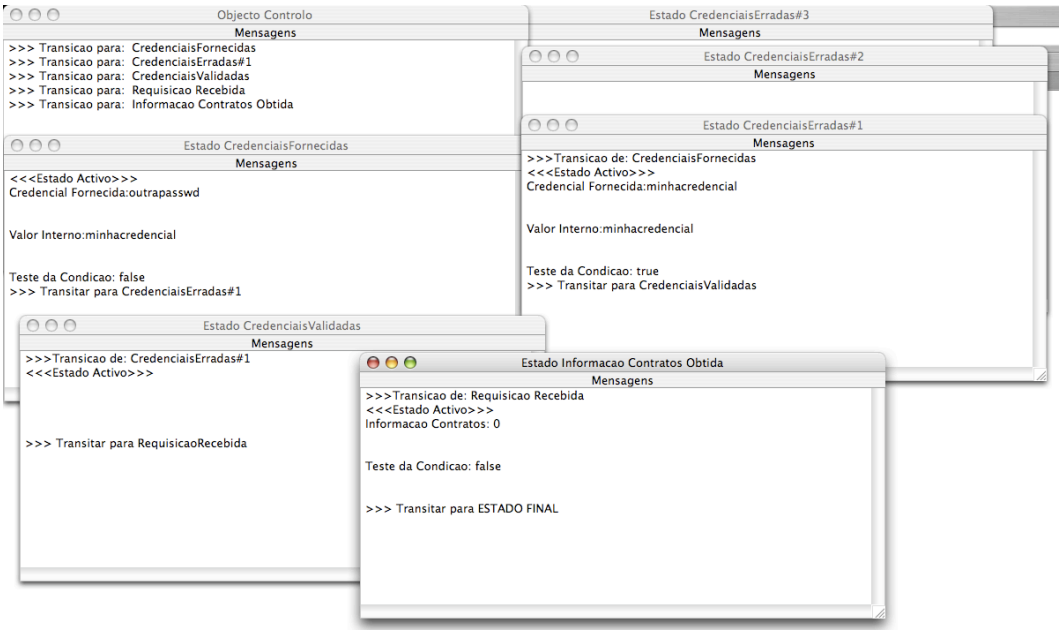


Figura 6.20: Verificação operacional da inexistência de contratos.

a execução do cenário, o que significa que o sistema fica num estado incoerente;

- resultado positivo - quando o cenário de teste é concluído na totalidade o que implica que todos os testes que foram efectuados foram verdadeiros, e

- resultado inconclusivo - quando a falha na avaliação de uma pré-condição impede que o resto do cenário seja testado. Esta situação pode ter ocorrido porque num dos estados que constituem o diagrama não ter sido respeitada uma condição.

A última situação pode ter sido alcançada porque não fazer sentido a execução do caso de uso continuar quando uma das pré-condições de um passo do caso de uso não tiver sido respeitada. Nesses casos o cliente pode validar o resultado por ter injectado valores que à partida se conhecia que não conduziriam a um resultado satisfatório, ou o cliente pode detectar uma não-conformidade na descrição comportamental que está a ser animada. Neste último caso é necessário voltar a iterar a especificação do caso de uso para a corrigir.

A necessidade de iterar a especificação é sempre causadora de impacto, porque implica voltar a fazer os artefactos de modelação que foram construídas. Num processo de modelação coeso a mudança numa peça de modelação implica sempre alterações nas outras que dela dependem. No caso do modelo de processo proposto uma mudança em

qualquer das três vistas - descrição dos casos de uso, diagrama de estados e diagrama de sequência - implica mudança nas outras.

No entanto, devido ao modo como os diagramas comportamentais são construídos existem condições objectivas que permitem minimizar o impacto das alterações. Note-se que quando um cliente detecta uma situação de excepção (não prevista) na validação operacional do caso de uso, não é a execução do caso de uso que estava a ser testada, mas sim a execução de um cenário desse mesmo caso de uso [Larman 05]. O cliente experimenta cenários do caso de uso e não a execução do caso de uso completo, pelo que este facto introduz um corte nos modelos que é necessário alterar.

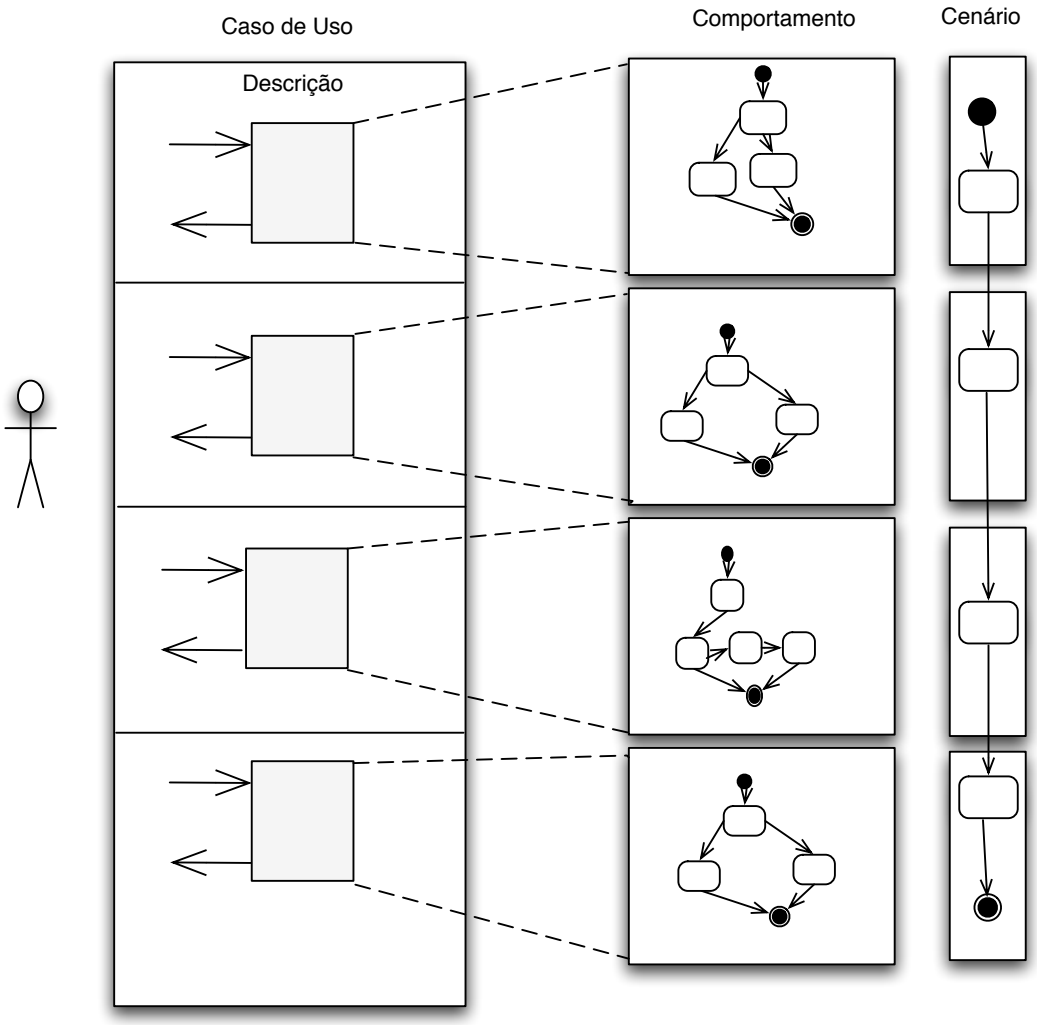


Figura 6.21: Transformação de um caso de uso na sua expressão de comportamento.

A Figura 6.21 apresenta graficamente o processo de construção da expressão de comportamento de um caso de uso a partir da sua narrativa textual, e ajuda a compreender o impacto que uma iteração provoca na especificação de um caso de uso.

Quando o caso de uso é animado, isto é, quando é animada a expressão do seu comportamento segundo o diagrama de estados, a ocorrência de uma situação não prevista acontece sempre num contexto conhecido que é um estado. Esse estado pertence à descrição de comportamento de um passo do caso de uso, como descrito na narrativa textual. Desta forma é possível delimitar imediatamente a zona do diagrama onde a situação de excepção foi identificada e verificar porque é que ela aconteceu.

Para analisar a causa da excepção é necessário verificar as expressões OCL que determinam a execução do estado em causa. Como cada uma dos passos do caso de uso tem associada pré e pós-condições que regulam a execução da sub-máquina de estados que o descreve, é necessário que a equipa de projecto analise, e corrija, essas expressões. As alterações a serem efectuadas às expressões OCL e eventualmente às guardas das transições devem ser reflectidas no diagrama de sequência do cenário que estava a ser testado.

Com o processo de construção das expressões de comportamento ilustrado na imagem, correcções ou alterações dos requisitos são feitas sabendo que a unidade de modularidade é um passo de um caso de uso. É possível mudar um passo de um caso de uso através da instanciação de uma nova sub-máquina de estados e da correspondente alteração no diagrama de sequência. As alterações no diagrama de sequência são circunstanciais e incidem nas mensagens que correspondem às acções expressas nas transições da nova sub-máquina de estados.

O impacto que uma iteração provoca é ainda minimizado pela utilização de OCL, porque a descrição das condições em que cada estado pode ser percorrido possibilita que se identifique qual é a expressão que não está a ser correctamente avaliada. As expressões OCL descrevem a lógica do negócio ao estabelecerem em cada um dos elementos do diagrama de estado as condições que permitem testar se estas estão a ser cumpridas. Mudanças aos requisitos que envolvam a alteração das regras de negócio são também concretizadas nas expressões OCL, sendo que é possível que o impacto de alterações se manifeste apenas a esse nível. A execução de um caso de uso, ou melhor de um dos seus cenários, pode apresentar um comportamento observável distinto apenas pela mudança das expressões OCL e mantendo a topologia do diagrama inalterada.

6.3.7 Outros protocolos de interligação dos estados

Um outro aspecto que o utilizador pode validar, tem a ver com o modelo de protocolo que regula a execução dos diagramas, isto é, do protótipo que os anima. Olhando para a natureza do problema do sistema de comércio electrónico, é expectável que a construção do sistema tenha que se preocupar com aspectos que uma pura análise funcional não contemple. Sendo o sistema composto por vários sub-sistemas, ou componentes, que podem estar localizados em diferentes ambientes de execução, a concretização dos casos de uso fica dependente do comportamento de todas as peças envolvidas. O caso de uso anteriormente detalhado, "Colocar Requisição", pode não ter uma execução que derive simplesmente da análise do diagrama de estado e diagramas de sequência associados, mas a própria configuração pretendida para o sistema pode influenciar a forma como devem ser interpretados os elementos de análise. Na realidade, nos elementos de modelação apresentados não é evidente que algumas das transições podem estar sujeitas a condicionalismos que advém da configuração e topologia do ambiente onde o sistema vai funcionar. Se não existir nenhuma anotação, ou um outro elemento de modelação, pode não ser evidente a quem analisa os diagramas qual é o modelo de comportamento do que se está a especificar. Esta falta de capacidade de transmitir informação relevante nos diagramas, sobre a topologia do sistema, lineariza em demasia os modelos produzidos e pode não permitir adquirir informação suficiente sobre a complexidade da descrição.

No caso do exemplo do sistema de comércio electrónico, suponha-se que o sistema que faz a gestão dos contratos é um sistema remoto para o qual o estabelecimento de um diálogo numa lógica síncrona (de *rendez vous*) pode não fazer sentido. Considere-se que esse sub-sistema aceita pedidos de confirmação da existência dos contratos, mas não responde imediatamente aos pedidos, mas possui uma fila de espera na qual esses pedidos são colocados e respondidos de acordo com a disponibilidade do sistema. Na execução do protótipo não seria compatível com a natureza do sistema a especificar, se não se tivessem em linha de conta estes requisitos, pelo que significam para a validação das especificações produzidas. No diagrama de estados a transição de "Requisição Recebida" para "Informação Contratos Obtida" pode ser modelada de forma a ser mais realista a validação operacional. Numa perspectiva de construção do protótipo, as mudanças são mínimas, havendo apenas que considerar mudar o tipo de diálogo que pode ser efectuado entre os dois objectos estado. Para criar um canal entre as duas entidades, bastaria declarar o canal como assíncrono, utilizando para tal a sintaxe

```
Canal reqRecebida_para_InfContrObtida = new Canal(new ComAssinc());
```

, em que a parametrização do construtor de `Canal` é efectuada através da criação de uma instância de `ComAssinc`. A mudança do tipo de comportamento associado ao canal, é transparente para quem utiliza os objectos na construção do protótipo, na medida em que do ponto de vista do conjunto de métodos disponibilizados não existem diferenças na sua assinatura. A concretização dos métodos de escrita (`write`) e leitura (`read`) é que reflecte o comportamento que se espera obter.

Operacionalização em Concorrência

Apesar de na maioria das situações se utilizar a animação do diagrama de estado com o intuito de verificar da correcção da especificação do comportamento, para uma interacção actor-sistema, é possível utilizar a animação de forma mais abrangente e permitir simular a execução da invocação do mesmo caso de uso, por parte de vários actores. Se o sistema consistir numa aplicação multi-utilizador, podem existir casos de uso cujo comportamento depende de interacções anteriores de outros actores com o sistema. Um exemplo de uma situação destas corresponde ao processo de encomenda de produtos em que a concorrência de pedidos de vários actores, a invocarem em simultâneo o mesmo caso de uso, origina que as acções a executar em cada estado dependam das interacções associadas às execuções que estão a decorrer em concorrência. Por exemplo pode, em determinado instante, não ser possível encomendar um produto que acabou de ser cativado (ou comprado) por um outro actor. Do ponto de vista da arquitectura de suporte, existe a capacidade de prototipar este tipo de comportamento, com os métodos de acesso aos canais a prevenirem eventuais situações de bloqueio (*deadlock*). No entanto, o acesso e verificação das condições associadas às transições, ou à permanência num determinado estado, assume contornos de maior complexidade na medida em que a avaliação depende de qual é o conjunto de dados (variáveis) que devem ser testadas, isto é, depende de qual é o actor que está associado aquele teste. É possível equacionar várias formas de resolver este problema, sendo que a procura de um modelo arquitectural que permita capturar e gerir a informação dos vários actores, de forma simples e modular, é tema de trabalho futuro.

6.4 O Sistema de Documentação da Universidade

O exemplo do sistema de comércio electrónico, representa um sistema constituído por diversos componentes autónomos e potencialmente distribuídos. Essa característica torna-o do ponto de vista da análise um sistema complexo, na medida em que além

dos requisitos puramente funcionais é necessário perceber qual é o impacto que a sua configuração lógica introduz na criação dos documentos de análise. É um sistema adequado para ser especificado de acordo com o processo proposto, na medida em que a possibilidade de se especificarem comportamentos dependentes da configuração do sistema é um contributo importante para a completude das tarefas de análise.

Um outro exemplo ao qual se aplicou também o processo proposto, é o do Serviço de Documentação da Biblioteca Geral da Universidade do Minho. A especificação deste sistema parte de um exemplo que foi fornecido aos alunos de 2º ciclo das unidades curriculares de especialização relacionadas com a especificação e modelação de sistemas software, onde se pretende demonstrar que o investimento numa fase inicial de construção completa e coerente dos documentos de análise, é determinante para a qualidade do produto final e para a correcta implementação do processo de construção.

O exemplo da biblioteca corresponde a um padrão de problemas que arquitecturalmente se descrevem de modo não muito complexo, mas onde o que é determinante para a correcta construção do sistema é a especificação comportamental dos casos de uso, na medida em que as regras de negócio são facilmente extraídas das expressões de comportamento destes. No exemplo foi considerada uma simplificação das entidades e funcionalidades de forma a ser mais sucinta a sua apresentação, não se considerando a componente de gestão do sistema (o *backoffice*), nem a integração de outros sistemas com os quais é normal que a biblioteca tenha mecanismos de integração aplicacional.

6.4.1 Regras do Serviço de Documentação

As regras de funcionamento da biblioteca estão documentadas em circular que regula os diversos aspectos relativos à prestação dos seus serviços. Considere-se o seguinte extracto do documento que descreve os principais requisitos do sistema e onde está também expressa a identificação das principais entidades que dele fazem parte.

> O sistema de empréstimo de obras dos SDUM (Serviços de Documentação da Universidade do Minho) é direito dos alunos, docentes, investigadores e funcionários da Universidade do Minho, os quais são designados por leitores. Pode ser requisitado todo o tipo de publicações excepto obras de referência, material audiovisual e algumas obras sinalizadas caso a caso. Para requisitar, o leitor, deve recolher a publicação na estante e dirigir-se ao balcão de atendimento da biblioteca, apresentando o seu cartão de leitor. Um leitor pode requisitar até um máximo de 6 obras e cada uma delas só pode estar em sua posse por 15 dias.

> Sempre que um leitor pretender o empréstimo de uma obra, e só uma, que esteja requisitada pode solicitar a sua reserva no balcão de atendimento, por telefone ou pela Internet.
>
> O período de requisição de publicações pode ser renovado. Essa renovação pode ser feita no balcão da biblioteca, por telefone ou através da Internet. Caso a publicação para a qual se pretende efectuar a renovação esteja reservada não é possível efectuar a renovação.
>
> A devolução das publicações tem que ser feita no balcão de atendimento da biblioteca. Caso essa devolução seja efectuada fora do prazo de requisição está sujeita a uma multa. O leitor deverá pagar 0,50 EUR por cada dia em atraso, e perde o direito a requisitar outras publicações enquanto não entregar a publicação em causa.

A descrição apresentada permite identificar os principais sub-sistemas existentes, bem como as principais entidades do sistema, mesmo que só se considere o extracto que foi transcrito do regulamento. No que diz respeito ao modo de utilização estão considerados o sistema de atendimento presencial em que o funcionário da biblioteca é o interlocutor do utilizador e o sistema de acesso via interface Web. Considere-se que o acesso às funcionalidades via telefone é semelhante ao modo presencial, sendo que em ambos o funcionário é que interage com o sistema, representando o utilizador. Em relação aos requisitos funcionais, é pretendido que o sistema disponibilize a requisição de obras, a renovação dos itens previamente requisitados e que faça a gestão relativa a possíveis atrasos na devolução e proceda à emissão de multas, de acordo com o tempo decorrido desde a data limite de entrega. Além destas funcionalidades, um leitor pode pedir para reservar uma obra que esteja no momento requisitada.

Com base na descrição fornecida as entidades principais do sistema são as seguintes:

- Leitor - que representa os utilizadores da biblioteca, que requisitam obras e renovam o seu prazo de entrega, bem como efectuam reservas de obras requisitadas;
- Obra - que sintetiza toda a informação relativa a uma publicação que é disponibilizada pela biblioteca;
- Requisição - que parece ser a entidade mais rica do sistema, na medida em que o funcionamento da biblioteca é assente sobretudo na criação deste tipo de entidades, de forma a registar os empréstimos de obras aos leitores, e
- Reserva - que permite registar a intenção de um leitor requisitar uma obra que está no momento indisponível.

Da leitura do extracto do regulamento também é possível identificar a necessidade da existência de mais entidades, a maior parte delas utilitárias, como sejam os registos relativos aos pagamentos, a criação de talões de requisição e a definição de entidades quer permitam efectuar cálculos baseados em diferenças temporais.

6.4.2 Diagrama de Casos de Uso

Tendo em conta a descrição do sistema como apresentada anteriormente, a elaboração do diagrama de casos de uso é relativamente fácil de obter. Considere-se que o sistema pode ser dividido em duas vistas principais, a que assegura o atendimento ao público, e que permite aos leitores efectuarem a requisição de obras que recolheram das estantes da biblioteca e a vista que especifica o sub-sistema que possibilita o acesso remoto aos serviços de documentação.

Na Figura 6.22, é apresentado o diagrama de casos de uso para as funcionalidades identificadas e que possibilitam a operação no espaço físico da biblioteca.

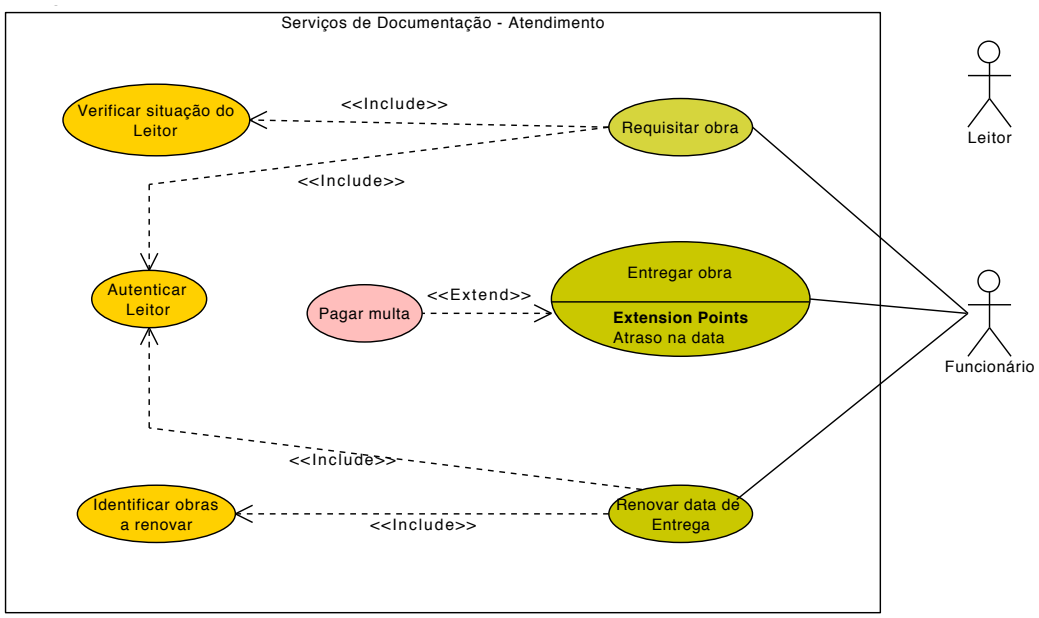


Figura 6.22: Diagrama de casos de uso para os Serviços de Documentação.

O diagrama apresentado identifica como actores o Leitor e o Funcionário, embora apenas o Funcionário é que interage com os casos de uso que o sistema disponibiliza. Embora do ponto de vista da construção do diagrama fosse possível existir ligação do Leitor para os casos de uso necessários, esta forma de representação implica que o leitor

é parte interveniente do processo de utilização do sistema software, mas quem executa as operações (no modo presencial) é o actor Funcionário.

O diagrama mostra já uma estruturação dos casos de uso, de modo a identificar os casos de uso que são utilizados no contexto de outros, mais abrangentes, e também o que acontece em situações de excepção. A funcionalidade do caso de uso "Autenticar Leitor" é utilizada pelos outros casos de uso, daí fazer todo o sentido que este possa ser reutilizado através do mecanismo de $<< include >>$. Também como casos de uso incluídos estão definidos "Verificar Situação do Leitor" e "Identificar Obras a Renovar", que são pedaços de funcionalidade que não são invocados por nenhum actor, mas que constituem um corpo funcional estanque e coerente. O diagrama apresenta também um caso de uso, "Pagar Multa", que é invocado caso tenha ocorrido uma situação excepcional no caso de uso "Entregar Obra". Essa situação excepcional corresponde à data da entrega ser posterior à que deve constar da requisição de empréstimo.

Além desta vista do diagrama de casos de uso apresenta-se, por uma questão de completude, o diagrama relativo às funcionalidades decorrentes do acesso aos serviços de documentação através da Internet. A Figura 6.23 mostra os casos de uso da vista da aplicação quando acedida remotamente. Neste tipo de acesso o leitor apenas poderá pesquisar obras ou reservar uma obra que tenha sido requisitada. Estas funcionalidades necessitam que o leitor esteja devidamente autenticado, pelo que o caso de uso "Autenticar Leitor" é incluído.

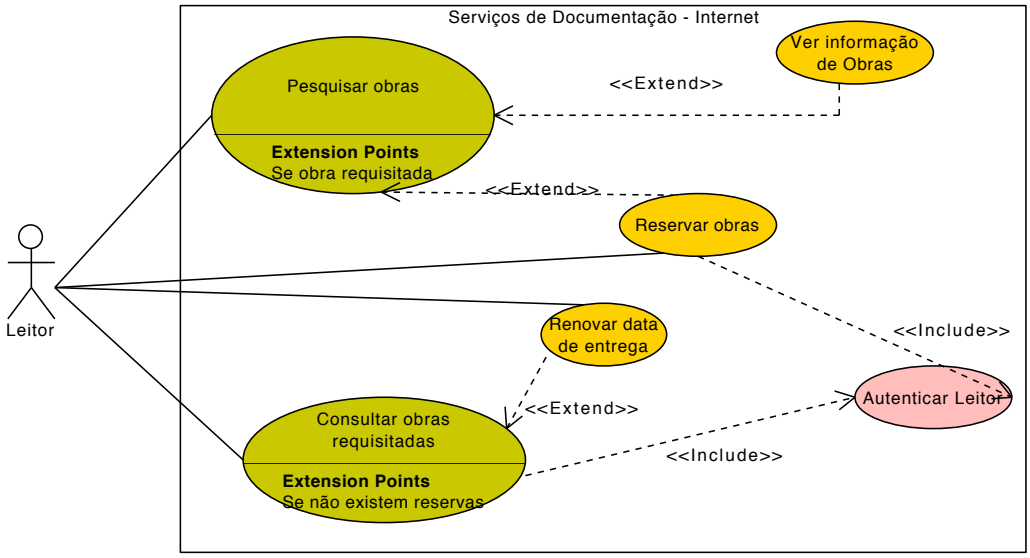


Figura 6.23: Casos de uso para a vista do sub-sistema de acesso remoto.

Os dois diagramas apresentados identificam os requisitos que estavam explícitos na descrição fornecida e através de uma análise funcional do problema, permitiram estruturar os casos de uso de forma a identificarem-se aqueles que são incluídos, e aqueles que são invocados em situação de excepção.

6.4.3 Modelo de Domínio

A Figura 6.24 apresenta o diagrama simplificado do modelo de domínio. Nele estão expressas as principais entidades do sistema, sendo que se optou por colocar a entidade Funcionário no diagrama, com o intuito de demonstrar que existem papéis distintos entre um Funcionário e um Leitor. É possível constatar no modelo de domínio que a relação entre Obra e Requisição expressa as correspondências existentes num determinado momento de funcionamento do sistema, isto é, uma obra apenas possui num determinado instante zero (nenhuma) ou uma requisição. Deverão existir no diagrama de classes final do sistema, outras classes que permitam endereçar as várias requisições que foram efectuadas para as obras existentes, possibilitando, por exemplo, a listagem das requisições de uma dada obra.

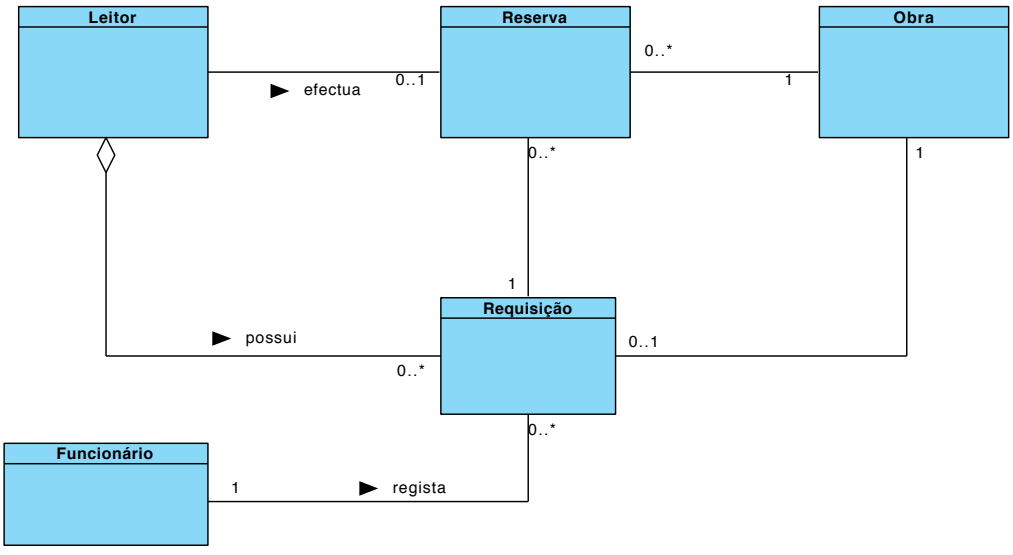


Figura 6.24: Modelo de Domínio simplificado para o Serviço de Documentação.

O diagrama de classes final só é obtido após a fase de análise, com os contributos desta, e é sujeito a princípios de boas práticas no que concerne à arquitectura das soluções e a outros aspectos tecnológicos. A fase de análise contribui para a elaboração

do diagrama de classes do sistema com a identificação iterativa de elementos (objectos, métodos e atributos) que farão parte do modelo da aplicação. Significa isso que no que concerne à identificação das operações de cada classe é possível obtê-las através do processo de construção dos diagramas comportamentais dos casos de uso, visto que as mensagens que são visíveis nos diagramas de estado e sequência vão corresponder a métodos dessas classes.

Embora nesta fase da análise do sistema não seja possível identificar com clareza as operações envolvidas, é possível detalhar um pouco mais o diagrama apresentado anteriormente e acrescentar informação que com o conhecimento que existe do sistema, se pode afirmar com um grau de certeza elevado que pertence à solução arquitectural final. A Figura 6.25, apresenta as entidades já identificadas e explicita com mais detalhe cada uma delas, possibilitando também uma melhor definição das associações existentes.

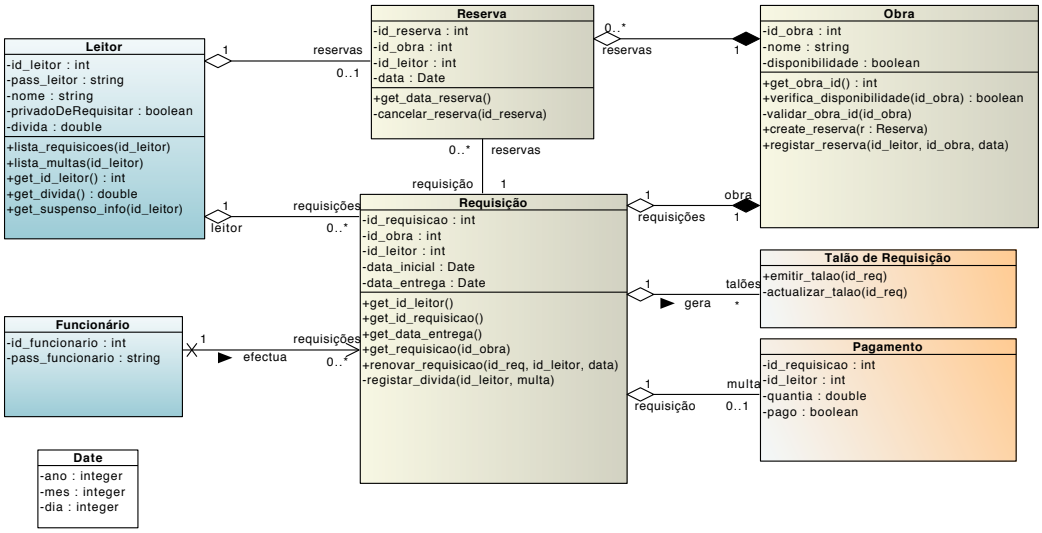


Figura 6.25: Modelo de Domínio mais detalhado para o Serviço de Documentação.

Note-se que entre as entidades Obra e Reserva e entre Obra e Requisição existem associações que podem ser navegadas através de qualquer um dos extremos. Significa que é possível a partir de cada das entidades encontrar informação da entidade associada, isto é, é possível dada uma Obra encontrar as reservas que foram feitas para ela e também a partir da Reserva determinar a Obra em questão. De modo a se saber quem é a entidade dona da associação, utilizou-se uma composição para estabelecer essa condição. É possível que no diagrama de classes da fase de concepção estes relacionamentos bidireccionais venham a desaparecer, por tornarem o modelo

muito complexo e excessivamente ligado, mas na fase de análise permitem estabelecer as relações semânticas que existem entre as entidades.

6.4.4 Caso de Uso "Requisitar Obra"

Tendo apresentado o diagrama de casos de uso, bem assim como o modelo de domínio onde se descrevem as principais entidades, relações e estado, é possível aplicar o processo aos casos de uso de forma a lhes especificar o comportamento. Seja o caso de uso "Requisitar Obra" que corresponde às actividades que modelam a procura de uma obra na biblioteca e a correspondente entrega ao funcionário para que ele conduza o processo de conclusão da requisição.

A descrição do caso de uso "Requisitar Obra" deve prever que não seja possível ao leitor a requisição de um livro se a sua situação face aos Serviços de Documentação for de incumprimento. A descrição textual, no arquétipo utilizado para textualmente identificar os passos importantes da execução, é apresentada na Figura 6.26.

Tendo em conta a descrição informal do caso de uso, como apresentada no formato que expressa os vários passos de execução e tendo em atenção os diferentes cenários alternativos que foram identificados, é possível ter uma ideia do controlo da execução do caso de uso. Os casos de uso incluídos podem ser detalhados nos diagramas que especificam o seu comportamento, através da descrição destes nos diagramas de estado e de sequência. Tal como foi feito para o exemplo do sistema de comércio electrónico, poderiam ser criados os diagramas de estado para cada um dos casos de uso incluídos e posteriormente adicionados ao diagrama de estado de "Requisitar Obra", como sendo uma sub-máquina de estados. No entanto, tendo em conta que a sua complexidade em termos comportamentais não é elevada, optou-se por ao nível do diagrama de estados ter todos os estados ao mesmo nível e fazer reflectir a estruturação modular no diagrama de sequência, de modo a que este não fique desnecessariamente complexo e seja possível identificar dessa forma os principais cenários do caso de uso.

Diagramas de sequência dos casos de uso incluídos

O caso de uso "Autenticar Leitor" é desencadeado pelo actor Funcionário que interage com o sistema. Por uma questão de simplificação considere-se que os actores interagem com uma entidade, Sistema UserInterface, que é a classe principal do sistema que coordena o controlo de diálogo interactivo e efectua a gestão da camada de apresentação. Poderíamos ter optado, tal como em [Gomaa 00], por colocar o estereótipo

Caso de Uso	Requisitar Obra
Sumário	O funcionário interage com o sistema com o intuito de requisitar a obra que o leitor apresenta ao balcão
Actor	Funcionário
Dependências	Casos de uso "Autenticar Leitor" e "Verificar Situação do Leitor" devem estar incluídos
Pré-condição	A aplicação está em inactividade, exibindo a janela inicial de escolha da opção escolhida
Descrição	1. Caso de uso "Autenticar Leitor" é invocado 2. Caso de uso "Verificar Situação do Leitor" é invocado 3. O funcionário insere o código da obra a requisitar 4. O sistema valida o código da obra 5. O sistema desbloqueia a obra para que esta possa ser requisitada 6. O funcionário regista o estado físico da obra 7. O sistema cria o registo da requisição 8. O sistema actualiza a ficha do leitor 9. O sistema actualiza o registo da obra 10. O sistema emite um talão de requisição da obra
Fluxos Alternativos	1. Se a autenticação não for bem sucedida, o sistema informa o funcionário e exibe a janela inicial 2. Se não for permitido ao leitor requisitar a obra, este é informado e o sistema exibe a janela principal 6. Se o código da obra for inválido, o sistema informa o leitor e funcionário
Pós-condição	Se o caso de uso teve sucesso, o sistema registou mais uma requisição e o estado da obra ficou alterado

Figura 6.26: Descrição do caso de uso "Requisitar Obra" do sistema dos Serviços de Documentação.

$<< userinterface >>$ de forma a explicitamente atribuir a esta classe o papel que lhe cabe no sistema. Após o envio da mensagem que inicializa a execução do caso de uso, o sistema deve de alguma forma ter de encontrar na sua colecção de leitores forma de validar se os dados de acesso são válidos. Não se conhecendo como é que vai ser construído o sistema, considerou-se que existe uma lista de leitores, designada por Leitores e que, nesta fase inicial, se considera que é do tipo `List<Leitor>`, independentemente da forma como vier a ser concretizada.

O diagrama de sequência para o caso de uso "Autenticar Leitor" é apresentado na Figura 6.27.

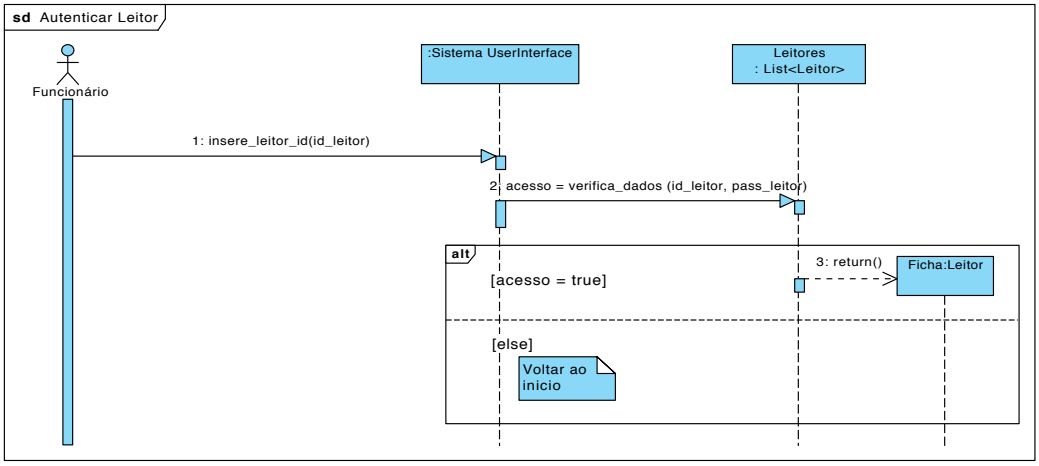


Figura 6.27: Diagrama de Sequência para "Autenticar Leitor".

No caso de ser possível identificar e autenticar o leitor, é devolvido o registo, Ficha, que tem a informação necessária sobre o utilizador da biblioteca.

Para a especificação do caso de uso "Verificar Situação do Leitor", a especificação de comportamento associado determina que se interrogue a Ficha do leitor de modo a determinar se existem algumas irregularidades relativas a multas ou à impossibilidade de requisitar mais obras. A especificação de comportamento deste caso de uso é relativamente simples e está expressa na Figura 6.28.

Diagrama de Estados

A especificação de comportamento sob a forma de um diagrama de estados para o caso de uso "Requisitar Obra", parte da especificação textual do caso de uso e detalha-a de modo a identificar qual é a execução do seu ciclo de vida. De acordo com o que foi

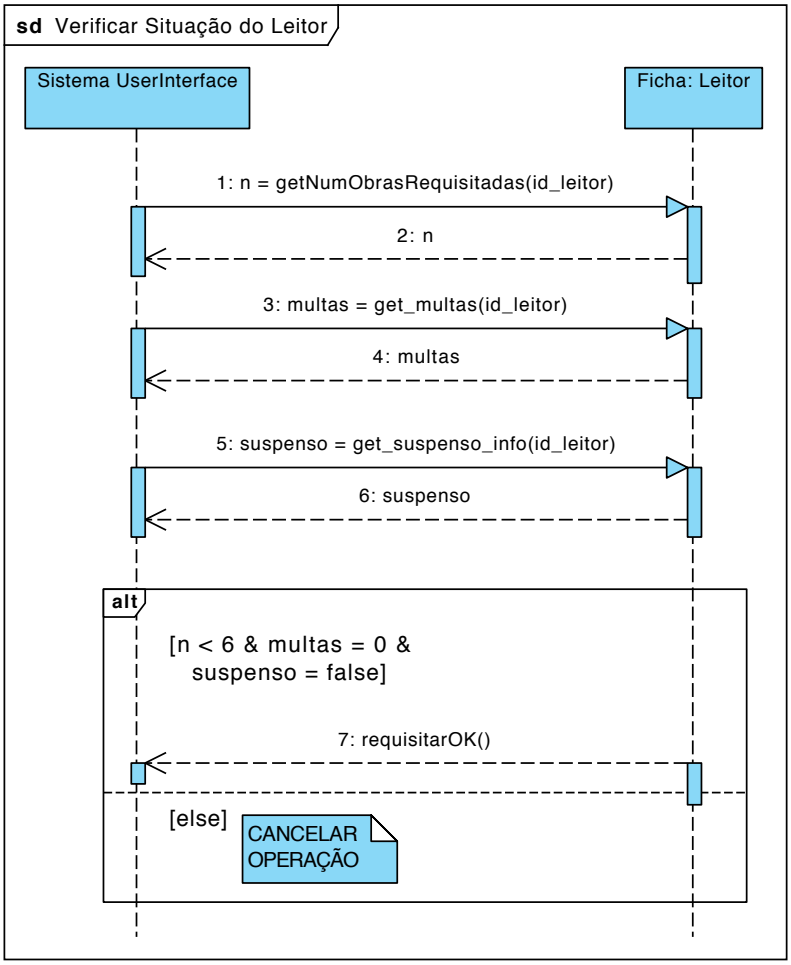


Figura 6.28: Diagrama de Sequência para "Verificar Situação do Leitor".

defendido quando se apresentou o processo proposto, cada linha da descrição textual do caso de uso dará origem a um estado na máquina de estados. No entanto, é possível optimizar o processo e juntar alguns desses estados de modo a tornar mais operacional a descrição do diagrama de estados.

A Figura 6.29 apresenta o diagrama de estados que descreve o modelo de execução do caso de uso, tal como apresentado inicialmente na sua descrição textual. Os dois primeiros estados que são percorridos Leitor Identificado e Acesso Leitor Efectuado, correspondem à execução do comportamento do caso de uso incluído "Autenticar Leitor", que se não obtiver a correcta identificação do leitor, termina a execução do caso de uso e envia informação para a camada de apresentação para sinalizar essa situação de excepção.

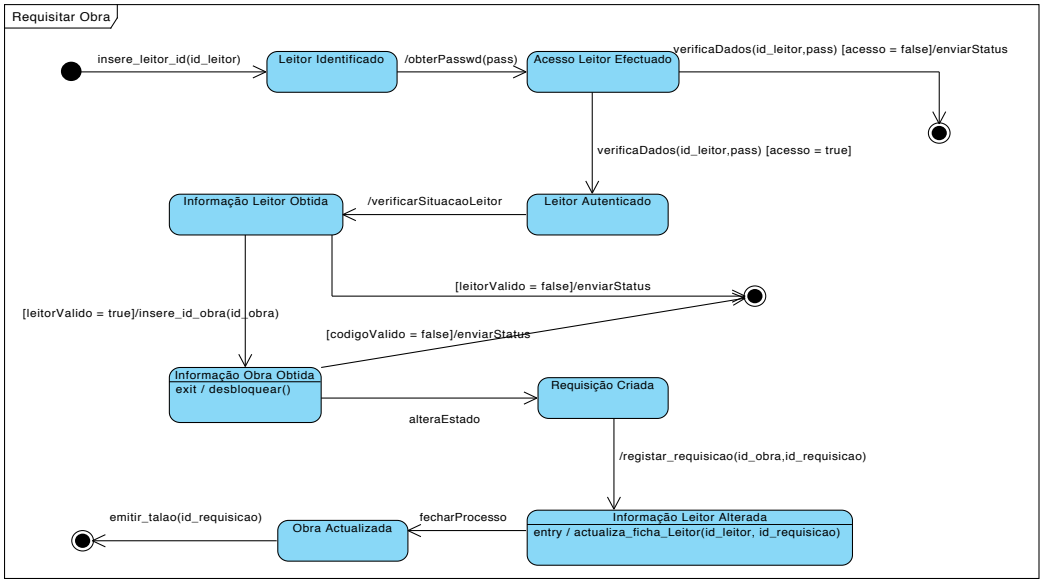


Figura 6.29: Diagrama de Estado do caso de uso "Requisitar Obra".

Na transição entre os estados Leitor Identificado e Acesso Leitor Efectuado é desencadeada a acção `obterPasswd`, que é invocada no contexto de uma transição automática. Depois de o leitor ter sido identificado pelo seu código, não é necessário lançar um evento para obter a *password*, sendo preferível nesta situação que a obtenção dessa informação esteja associada à acção `obterPasswd`. O mesmo critério foi utilizado na transição que se representa entre Leitor Autenticado e Informação Leitor Obtida, onde não é necessário esperar que surja um evento para que se proceda à invocação de um método que determine qual é o estado do leitor no sistema de documentação. Esta invocação, com o intuito de determinar se o leitor pode requisitar obras, corresponde à descrição do comportamento do caso de uso incluído "Verificar Situação do Leitor". Caso o leitor, por algum dos motivos especificados na interacção de "Verificar Situação do Leitor", não poder requisitar a obra, então é feita uma transição para o estado final e é enviada informação para a camada de apresentação. Da mesma forma, caso a obra solicitada, não exista, ou não possa ser requisitada, a execução do diagrama termina e o actor é informado da razão do término.

Como está explícito na descrição textual do caso de uso, a obra depois de identificada deve ser desbloqueada no sistema para que possa ser requisitada pelo leitor. Este procedimento ocorre automaticamente sempre que é identificada a obra e se pretende requisitá-la. Dessa forma é possível especificar que sempre que o controlo de execução transita do estado Informação Obra Obtida para o estado Requisição Criada, antes da

requisição ser criada é necessário que seja feito o desbloqueio da obra. Para tal no estado Informação Obra Obtida associa-se à actividade `exit`, a invocação da operação que desbloqueia a obra e representa-se no estado na forma:

```
exit/desbloquear()
```

Seguindo a mesma linha de raciocínio, no estado Informação Leitor Alterada, após se transitar para este estado por se ter efectuado o registo da requisição, que é a acção associada à transição de estado, é possível desencadear uma acção que actualize a informação do leitor através da inserção de uma nova requisição. Dessa forma ao entrar no estado, e associada à actividade `entry`, invoca-se o método que irá efectuar essa actualização. A expressão representa-se como:

```
entry/actualiza_ficha_Leitor(id_leitor, id_requisicao)
```

Por último, no estado Obra Actualizada, quando o controlo de execução do caso de uso chega a este ponto, é preciso que a informação relativa à obra seja actualizada. Note-se que para este processo ser desencadeado é necessário que o evento `fecharProcesso` seja enviado, o que provavelmente acontecerá através da camada de apresentação. A exemplo da descrição efectuada anteriormente para outros estados, também em Obra Actualizada, a invocação da mensagem que actualiza a informação da obra pode ser associada a `entry`, na forma

```
entry/actualiza_registo_obra(id_obra,id_requisicao)
```

Por último, antes de atingir o estado final, a execução do diagrama de estados efectua a emissão do talão, através da invocação de um método com esse propósito.

O diagrama de estados que foi apresentado para o caso de uso "Requisitar Obra", descreve o seu comportamento como este está expresso na descrição textual apresentada, que a equipa de projecto fez com o cliente. A transformação da descrição textual ao ser transposta para uma máquina de estados, permite que se efectue raciocínio formal sobre os fios de execução possíveis, mas também obriga a que se descrevam as acções associadas às transições. Dessa forma descobrem-se, à medida que se constrói o diagrama de estados, os métodos que são necessários à concretização do comportamento do caso de uso. Esses métodos são utilizados na descrição da interacção na vista do diagrama de sequência e alimentam o diagrama de classes do sistema. Na construção do diagrama de estados para o caso de uso "Requisitar Obra", foi identificada a necessidade da existência dos seguintes métodos:

- `verifica_dados(id_leitor,pass)` - para determinar a autenticação do leitor no sistema;
- `insere_id_obra(id_obra)` - que envia para o sistema a identificação da obra a requisitar;
- `desbloquear()` - que muda o estado da obra, permitindo que possa ser requisitada;
- `registar_requisicao(id_obra,id_requisicao)` - que adiciona uma nova requisição;
- `actualiza_ficha_Leitor(id_leitor,id_requisicao)` - que adiciona ao leitor a informação de mais uma requisição;
- `actualiza_registo_obra(id_obra,id_requisicao)` - que actualiza a informação da obra com o código da requisição que actualmente está activa, e
- `emitir_talao(id_requisicao)` - que origina informação (listagem) acerca dos detalhes da requisição.

A construção do diagrama de estados, possibilitou que esta informação fosse capturada, sendo que ela faz parte do património da fase de concepção arquitectural do sistema. O facto de ter sido capturada durante a fase de análise permite que a equipa de projecto chegue à fase de concepção com mais informação do que a disponibilizada por processos mais tradicionais e rígidos.

Diagrama de Sequência

A descrição do comportamento efectuada no diagrama de estados identifica os estados pelos quais passa a execução do caso de uso e detalha todos os possíveis cenários que se podem traçar a partir do caso de uso. O diagrama de sequência, descreve uma interacção e pode representar apenas um cenário. De acordo com o princípio da modularidade de representação dos diagramas de sequência, apresentado em 4.3.4, a descrição contém a estrutura que permite incluir referência a outras descrições de interacções que especificam os diferentes cenários. No diagrama de sequência de "Requisitar Obra", os casos de uso incluídos são representados através da referência a uma interacção que está descrita noutros diagramas. No caso dos diagramas de sequência para "Autenticar Leitor" e "Verificar Situação do Leitor", estes estão descritos nas Figuras 6.27 e 6.28.

A Figura 6.30 apresenta o diagrama de sequência que descreve o cenário normal do caso de uso. As mensagens que são trocadas entre as entidades correspondem a operações que resultaram da acção de especificação do comportamento ao nível do diagrama de estados.

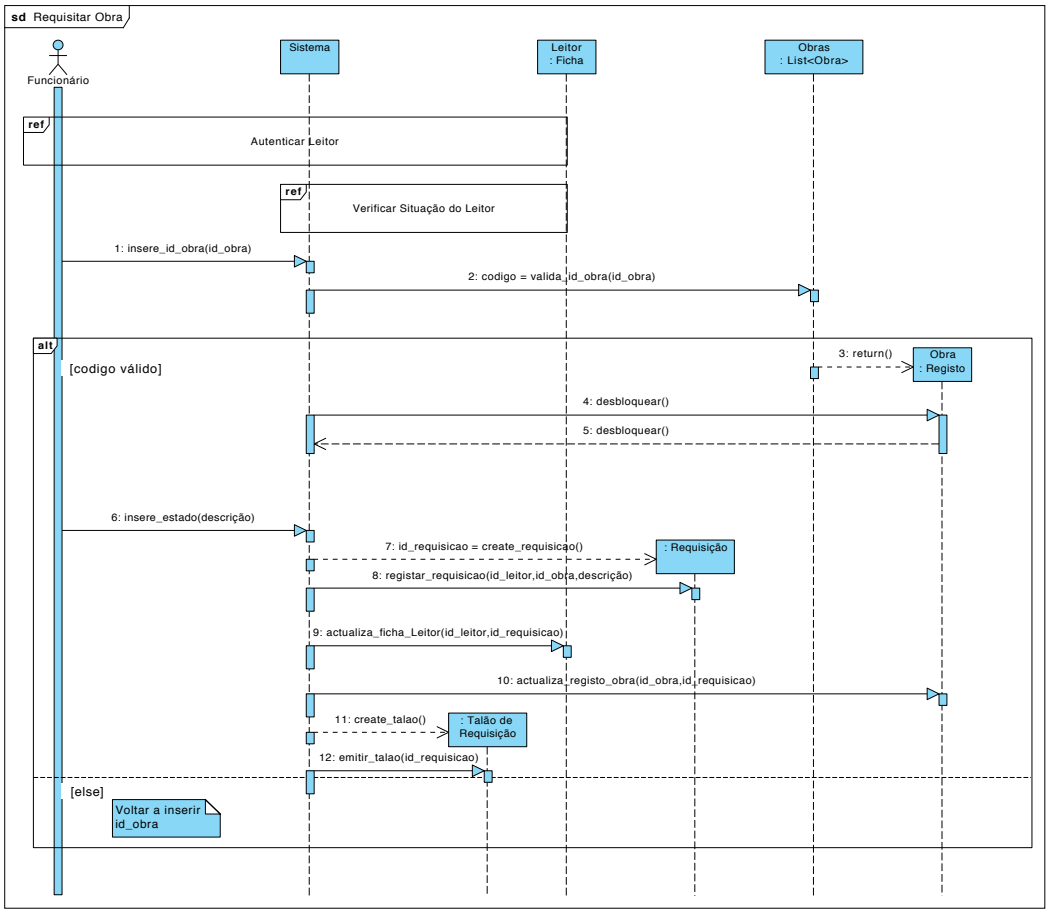


Figura 6.30: Diagrama de Sequência para "Requisitar Obra".

O diagrama de sequência deve concretizar em entidades do modelo do domínio os receptores das mensagens que são identificadas no diagrama de estados. No diagrama de sequência do caso de uso "Requisitar Obra", apenas se acrescentou por comodidade de representação uma entidade que contém todas as obras e que para efeitos do diagrama se considera ser do tipo `List<Obra>`. Caberá à equipa de projecto na fase da concepção, concretizar se é uma lista, ou outro tipo de estruturação, que se pretende utilizar.

Especificação em OCL

Depois de descrito o comportamento do caso de uso com recurso aos diagramas de estado e de sequência, a especificação em OCL de restrições ao funcionamento, acrescenta informação de carácter mais formal à especificação produzida.

Considerem-se primeiro as restrições que se podem considerar tendo em conta as entidades do modelo de domínio. As expressões que definem o invariante são:

- id_leitor maior que 0 e é um identificador único. A dívida é sempre positiva ou igual a 0.

```
context Leitor
inv : self.divida >= 0
inv : self.id_leitor > 0
inv : Leitor.allInstances -> forAll(p1, p2 | p1<>p2 implies
p1.id_leitor <> p2.id_leitor)
```

  Por uma questão de compreensão, e síntese, a última expressão pode ser escrita como

```
inv: Leitor.allInstances->isUnique(id_leitor)
```

- id_funcionario é maior que 0 e é um identificador único

```
context Funcionario
inv : self.id_funcionario > 0
inv : Funcionario.allInstances -> isUnique(id_funcionario)
```

- id_reserva maior que 0 e é um identificador único

```
context Reserva
inv : self.id_reserva > 0
inv : Reserva.allInstances -> isUnique(id_reserva)
```

- id_requisicao maior que 0 e é um identificador único

```
context Requisicao
inv : self.id_requisicao > 0
inv : Requisicao.allInstances -> isUnique(id_requisicao)
```

- id_obra maior que 0 e é um identificador único

```
context Obra
inv : self.id_obra > 0
inv : Obra.allInstances -> isUnique(id_obra)
```

- quantia maior que 0

```
context Pagamento
inv : self.quantia > 0
```

- um leitor só pode fazer no máximo 1 reserva e requisitar um máximo de 6 obras.

```
context Leitor
inv : (self.reservas->size() >=0) and (self.reservas->size() <2)
inv : (self.requisicoes->size() >= 0) and
(self.requisicoes->size() <= 6)
```

- um funcionário pode executar 0 ou mais requisições de obras.

```
context Funcionario
inv : (self.requisicoes->size() >= 0)
```

- cada requisição está associada a pelo menos um talão de requisição (entregue no momento da requisição). Outros talões podem ser gerados no caso de serem feitas renovações da data de entrega

```
context Requisicao
inv : (self.taloes->size() >= 1)
```

- uma obra pode ter várias reservas, mas no entanto, apenas uma requisição.

```
context Obra
inv : (self.requisicoes->size() >= 0) and
(self.requisicoes->size() <= 1)
inv : (self.reservas->size() >= 0)
```

- associada a uma requisição podem ser criadas várias reservas e uma reserva terá sempre uma requisição associada.

```
context Requisicao
inv : (self.reservas->size() >= 0)

context Reserva
inv : (self.requisicao->size() = 1)
```

A informação recolhida nas expressões do invariante, permite certificar a modelação feita ao nível do modelo de domínio. Estas expressões são fundamentalmente orientadas à regulação das associações entre entidades, sendo que com mais conhecimento do problema será possível à equipa de projecto, elaborar mais expressões que devam ser invariantes.

Interessa também especificar o que acontece no contexto das entidades quando são enviadas mensagens que as referem como destinatário. Para que seja possível escrever as expressões OCL é necessário referir atributos das entidades relacionadas, atributos esses que são alterados, ou testados, durante a execução do caso de uso. Uma vantagem da descrição do caso de uso no correspondente diagrama de estados, consiste na identificação da fase da execução em que esses atributos são alterados. A Figura 6.25 apresenta a definição do modelo de domínio já com a identificação dos atributos principais das entidades, pelo que serve de referência à escrita das expressões em OCL.

A mais que provável criação de uma entidade que agregue as diferentes entidades e que será visível no diagrama de classes que resultará da fase de concepção, originará mudanças tanto a nível das entidades representadas, dos seus relacionamentos, mas também ao nível do comportamento que se prevê possa existir. Na construção dos diagramas de estado e de sequência para os exemplos do caso de estudo, utilizam-se assinaturas de métodos que nesta fase da modelação pretendem ser elementos de documentação, apresentando os parâmetros que se julgam importantes. Essas declarações poderão ser alteradas posteriormente pela equipa de projecto, tendo em conta a natureza das tecnologias que estiverem a utilizar. Recomenda-se contudo que essas alterações sejam comunicadas e incorporadas novamente na documentação da fase de análise.

Considerando o modelo de domínio apresentado, podem ser definidas expressões OCL para algumas das operações referidas nos diagramas de estado e de sequência. No caso da criação de uma nova requisição para uma obra, a expressão OCL correspondente é:

```
context Obra::create_requisicao(r: Requisicao)
inv  : r.data_entrega - r.data_inicial = 15  -- tempo permitido de posse
pre  : self.disponibilidade = true           -- deve ser possível requisitar a obra
```

```
post : requisicoes = requisicoes@pre->including(r)
```

A operação `create_requisicao` recebe uma requisição e acrescenta-a às requisições já existentes. A expressão que a define tem impacto na descrição de comportamento do diagrama de estados, porque reforça as condições em que é possível efectuar as transições de estado, bem assim como condiciona a execução da interacção, tal como esta é expressa no diagrama de sequência. Ao nível do diagrama de estados, o estado Informação Obra Obtida deve testar como pré-condição, para que seja possível continuar com o processo de requisição, a disponibilidade da obra para ser efectivamente requisitada. Dessa forma a descrição do estado Informação Obra Obtida deve passar a prever que:

```
entry/disponibilidade = true
```

Seguindo a mesma linha de raciocínio, após a requisição ter sido criada, no estado Requisição Criada, deve ser adicionado como detalhe à descrição do estado a expressão:

```
exit/requisicoes = requisicoes@pre->including{r}
```

Esta expressão possibilita que previamente à efectivação da transição se teste se a nova colecção de requisições inclui, ou não, a nova requisição que foi criada.

A construção de expressões OCL deve ser feita para todas as operações que tenham sido invocadas na construção dos diagramas utilizados para descrever o caso de uso. Por exemplo, no caso de uso incluído "Verificar Situação do Leitor", tornou-se necessário pedir a uma entidade do tipo Leitor o número de requisições existentes para aquele leitor. A expressão OCL correspondente é:

```
context Leitor::getNumObrasRequisitadas():int
post: result = (Requisicao.allInstances->collect
                (r:Requisicao|r.id_leitor = self.leitor)->size()
```

6.4.5 Caso de Uso "Renovar Requisição"

De acordo com o processo de modelação proposto, idêntico procedimento deverá ser seguido para todos os casos de uso, de forma a especificar completamente o seu comportamento e contribuir para a recolha que possa ser útil às fases seguintes do processo.

Considere-se ainda o exemplo da funcionalidade que possibilita a renovação de uma requisição e que é disponibilizada através do caso de uso "Renovar Requisição". Este caso de uso é utilizado sempre que um leitor pretende prolongar no tempo a requisição que actualmente possui para uma determinada obra. Por princípio, essa renovação é possível, desde que não tenha sido efectuada uma reserva para essa obra, ou então que a requisição actual não tenha ultrapassado já a data limite de entrega.

Seguindo o processo definido, procede-se à criação da descrição textual, de acordo com o arquétipo definido para o efeito. A descrição textual é apresentada na Figura 6.31.

Recorrendo à mesma lógica de transformação da notação textual do caso de uso para os diagramas comportamentais, em que se transformam cada um dos passos da descrição num estado, apresentam-se nas próximas secções os diagramas de estado e sequência que se podem construir para detalhar o caso de uso.

A exemplo do que se passou no caso de uso anteriormente apresentado e detalhado, também na descrição de "Renovar Requisição" é necessário contar com dois casos de uso incluídos, "Autenticar Leitor" e "Identificar Obra a Renovar". Em relação ao primeiro, a especificação do seu comportamento já foi detalhada na Figura 6.27. Para o caso de uso "Identificar Obra a Renovar" a especificação da interacção que conduz à sua execução está expressa na Figura 6.32.

Diagrama de Estado

Como também se referiu aquando da análise do caso de uso "Requisitar Obra", o algoritmo de transformação das descrições textuais para o diagrama de estados pode ser optimizado, através da junção de alguns estados, com o objectivo de dotar o diagrama de uma maior capacidade de leitura.

A Figura 6.33 ilustra o diagrama de estados que descreve o modelo de execução do caso de uso, de acordo com a sua descrição. Os estados Leitor Identificado e Acesso Leitor Efectuado correspondem à execução do comportamento do caso de uso incluído "Autenticar Leitor", que na impossibilidade da identificação do leitor, termina a execução do caso de uso e sinaliza, via interface com o utilizador, essa situação de excepção.

O estado Informação Obra Obtida, assinala o momento em que é possível obter a informação sobre o estado de reservas associado a uma obra. A partir desse estado, o controlo de execução do diagrama de estados determina se é, ou não, possível prosseguir com o processo de renovação. Este estado pode ser descrito como tendo uma actividade

Caso de Uso	Renovar Requisição
Sumário	O funcionário interage com o sistema com o intuito de renovar a requisição para a obra que o leitor actualmente tem requisitada
Actor	Funcionário
Dependências	Casos de uso "Autenticar Leitor" e "Identificar Obra a Renovar" devem estar incluídos
Pré-condição	A form Web está em inactividade, exibindo a janela inicial de escolha da opção escolhida
Descrição	1. Caso de uso "Autenticar Leitor" é invocado 2. Caso de uso "Identificar Obra a Renovar" é invocado 3. O funcionário insere o código da obra a renovar 4. O sistema valida o código da obra 5. O sistema verifica a situação da obra 6. O sistema altera a data de entrega da obra incrementando-a em 15 dias 7. O sistema actualiza o registo da requisição 8. O sistema emite um talão de requisição da obra
Fluxos Alternativos	1. Se a autenticação não for bem sucedida, o sistema informa o funcionário e exibe a janela inicial 5a. Se não for permitido ao leitor requisitar a obra por esta se encontrar reservada, este é informado e o sistema exibe a janela principal 5b. Se a data de entrega da obra já foi ultrapassada então a renovação não pode ser efectuada e o sistema informa o leitor e funcionário
Pós-condição	Se o caso de uso teve sucesso, o sistema renovou a data de entrega da requisição para a respectiva obra

Figura 6.31: Descrição do caso de uso "Renovar Requisição" do sistema dos Serviços de Documentação.

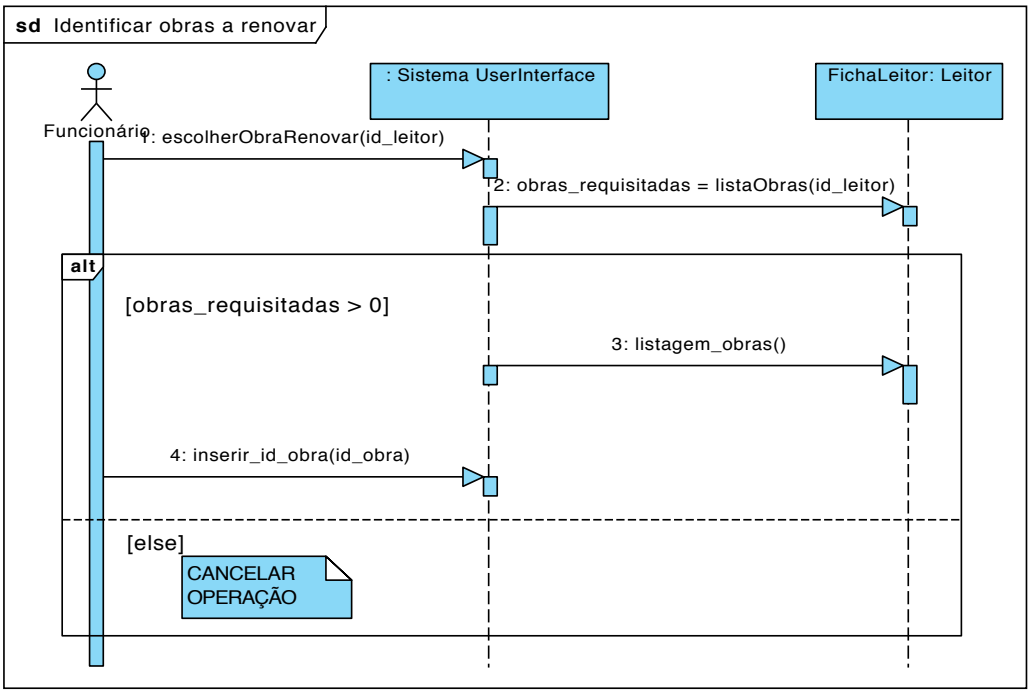


Figura 6.32: Diagrama de Sequência para o caso de uso "Identificar Obra a Renovar".

de entrada em que adquire a informação necessária sobre as reservas, de modo a ser posteriormente possível efectuar as transições de acordo com o resultado da operação invocada.A obtenção desse valor está dependente da invocação de uma operação que recolha essa informação, sendo que no caso da descrição do caso de uso isso é feito por `verifica_reservas(id_obra)`.

A actividade de entrada no estado é descrita como:

```
entry/numReservas = verifica_reservas(id_obra)
```

As transições que são efectuadas a partir deste estado são dependentes do teste da condição que verifica se o número de reservas é, ou não, igual a zero. No caso de ser igual a zero, significa que é possível prosseguir com o processo de renovação.

Importa fazer para este caso de uso o mesmo exercício realizado para o caso de uso anterior ("Requisitar Obra") e identificar os métodos descobertos na sua especificação de comportamento. Esses métodos são:

- `listaObras(id_leitor)` - que determina a quantidade de obras que o leitor tem actualmente requisitadas;

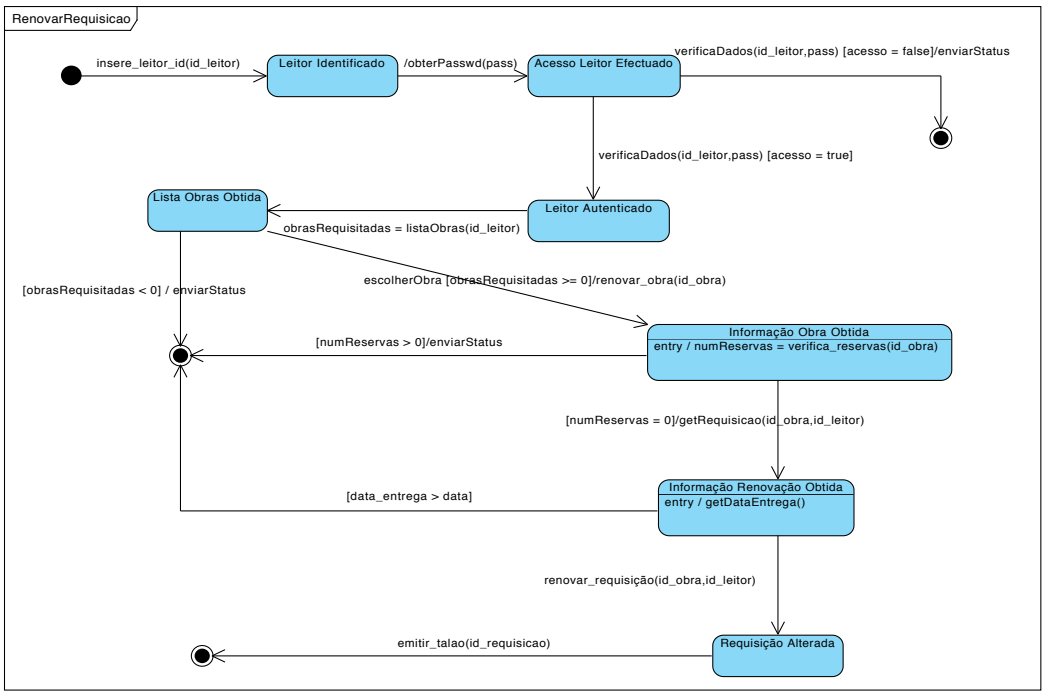


Figura 6.33: Diagrama de Estado do caso de uso "Renovar Requisição".

- `renovar_obra(id_obra)` - que é a operação que desencadeia a renovação de uma obra, depois de escolhida entre as obras que o leitor tem requisitadas;
- `verifica_reservas(id_obra)` - que determina as reservas para uma obra, e
- `renovar_requisicao(id_obra, id_leitor)` - que actualiza a requisição da obra em questão.

Excluem-se desta análise, as operações que apenas interrogam os objectos para obter informação sobre o seu estado interno, por estas aparecerem naturalmente na fase de desenvolvimento, sendo que muitos dos ambientes integrados de modelação e desenvolvimento já os disponibilizam de forma automática.

Diagrama de Sequência

O diagrama de sequência que é apresentado na Figura 6.34 ilustra a interacção que pode ser traçada a partir da descrição de comportamento efectuada no diagrama de estados.

A construção do diagrama de sequência além de utilizar as operações identificadas no diagrama de estados, tem por objectivo identificar quais são as entidades do modelo de domínio que estão envolvidas na interacção. Como em exemplos anteriores recorre-se à utilização de listas de objectos de um determinado tipo, como `List<Requisicao>`, para referenciar todos os objectos daquele tipo. Existirá seguramente na solução arquitectural final um objecto que agregue todos os outros e que faz a gestão de colecções de entidades de um determinado tipo, mas neste momento do projecto essa informação ainda não é conhecida e não é relevante.

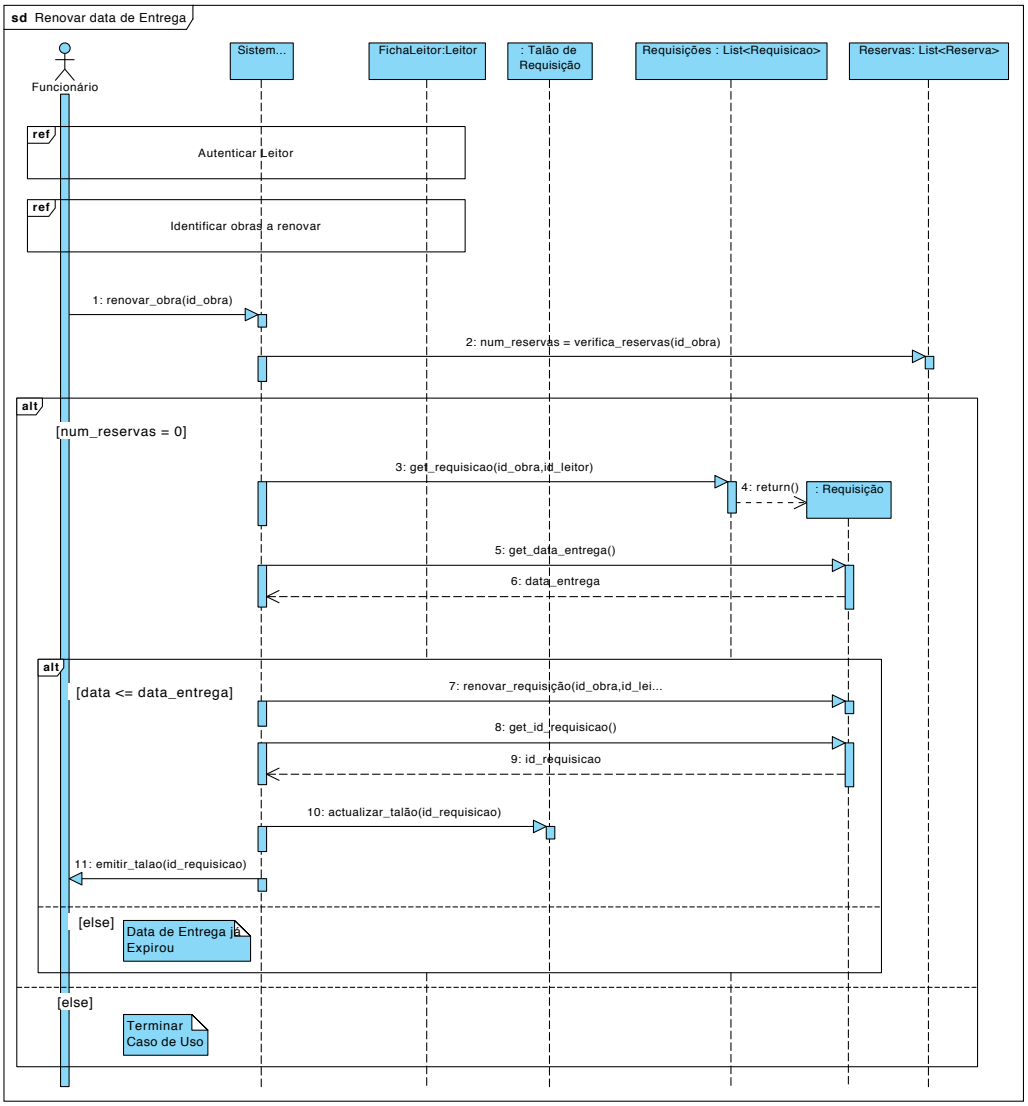


Figura 6.34: Diagrama de Sequência do caso de uso “Renovar Requisição”.

Especificação em OCL

Este caso de uso permite acrescentar mais expressões rigorosas, descritas em OCL, que constituem documentação do projecto, mas que podem ser também utilizadas para decorar os seus diagramas comportamentais.

Considerando o modelo de domínio apresentado, é possível definir em OCL a operação que efectua a renovação da requisição, como sendo:

```
context Requisicao::renovar_requisicao(id_obra:int,
                                       id_leitor:int,id_req:int)
inv  : self.leitor->select(l : Leitor |
                           l.id_leitor = self.id_leitor)->size() = 1
inv  : Requisicao.allInstances->exists(r: Requisicao |
                                       r.id_requisicao = id_req)
pre  : self.reservas->size() = 0            -- não existem reservas para a obra
pre  : self.data_entrega > dataSistema      -- seja a data actual
post : self.data_entrega =
            self.data_entrega@pre + 15      -- renovação por 15 dias
```

Por uma questão de simplificação na expressão anterior utilizaram-se datas como se fossem valores inteiros, de modo a ser fácil a sua comparação. Por uma questão de completude, dever-se-ia ter definido um tipo, `Date`, e nele ter definido as operações que permitem construir a representação de uma data e possibilitam a comparação de datas. Por não trazer valor acrescentado essa definição de um novo tipo de dados, optou-se por representar as datas como sendo valores inteiros e por assumir que a variável `dataSistema` representa a informação de calendário tal como é fornecida pelo sistema operativo que suporta a execução.

A informação que se retira da expressão `renovar_requisicao` pode ser utilizada para alimentar o diagrama de estados, possibilitando que no estado Requisição Alterada se possa acrescentar como actividade

```
entry/self.data_entrega = self.data_entrega@pre +15
```

Da mesma forma, nas transições onde se faz referência à variável `numReservas` e se efectua a sua comparação com zero, a condição na transição poderia ser substituída por `self.reservas->size() = 0` ou `not self.reservas->size() = 0`.

6.5 Avaliação da utilização da OCL

O processo que tem vindo a ser apresentado, tem como objectivo último a dotação de uma maior sustentação à fase de análise. Nessa perspectiva, a tarefa da equipa de projecto consiste em produzir os elementos de especificação necessários, para documentar, da melhor forma, os requisitos expressos na forma de casos de uso. Por um lado, este incremento de documentação e modelação faz-se através do detalhe multi-vista que o processo propõe, possibilitando a recolha sistemática e iterativa de informação. A construção dos diagramas de estado e de sequência permite que se detalhem as regras que definem a execução do caso de uso, incorporando nesses diagramas elementos que são descobertos e que passam a fazer parte do património do documento de análise. Entre esses elementos de especificação que são descobertos, três são particularmente importantes:

1. os objectos envolvidos, ou a definição das entidades que serão a matriz desses objectos. Quando se especificam as interacções (no diagrama de sequência) é necessário encontrar entidades que suportem as trocas de mensagens. Estas entidades, através de refinamento e iteração, constituirão as classes da aplicação. Note-se que nem todas estas classes são do mesmo tipo, ou pertencem à mesma camada (se estivermos a pensar numa lógica de aplicações multi-camada), existindo algumas que são pertencentes à camada de apresentação, e que são as receptoras iniciais da interacção, outras à camada de negócio e outras que são abstracções de entidades do repositório de dados;

2. as mensagens que são trocadas entre os objectos que fazem parte da interacção, ou que são desencadeadas aquando das transições de estado. A identificação dos métodos é feita de forma iterativa e é justificada pela necessidade de especificação do comportamento e não é um exercício feito posteriormente com base nas classes existentes, onde se procura determinar qual deve ser o conjunto de métodos necessário, e

3. as restrições, isto é, as condições booleanas, que é necessário garantir quer na especificação do comportamento com o diagrama de estado, quer na descrição da interacção. Essas condições são resultado da explicitação das regras de negócio e são colocadas nos diagramas de forma a garantirem que as descrições especificadas têm um grau de adesão à realidade elevado. Estas restrições podem ser enquadradas em dois grandes grupos: as (i) que são decorrentes da análise de comportamento e que estão tipicamente associadas às transições de estado, ao

envio de mensagens entre objectos e à alteração de variáveis do estado interno e (ii) as restrições que são resultado da estruturação de entidades que se define ao nível do diagrama de classe e que são reguladoras da forma como essas entidades se relacionam. Neste último caso, essas restrições podem ser expressas nos elementos que decoram as associações (em termos de multiplicidades, navegabilidade, entre outras), bem assim como na escrita de expressões que relacionem várias entidades.

Com a descoberta desta informação, o processo de análise torna-se mais rico, recolhe mais informação e assume um papel importante na passagem para a fase de concepção arquitectural, ao fornecer informação mais estruturada do que uma captura de requisitos mais informal garantiria. No entanto, a informação recolhida na fase de análise não é toda da mesma natureza. Existe informação mais informal, que descreve os casos de uso textualmente, embora complementada com a indicação de pré e pós-condições e invariante, e informação mais detalhada como a expressa pelos diagramas comportamentais. A restante informação que tem a ver com a regulação comportamental da execução do caso de uso, deve poder ser recolhida do modo mais formal possível. Como a OCL faz parte integrante da UML, não faria sentido não a utilizar com este propósito. A linguagem apresenta todas as características necessárias a ser utilizada neste tipo de especificação e apesar de não ser previsível que os utilizadores consigam ler as descrições efectuadas em OCL, esse não é um aspecto negativo da abordagem. A informação rigorosa que é recolhida em OCL é o resultado da transformação de conhecimento informal, ou disperso, que fica registado sob a forma de expressões que são semanticamente coerentes e possibilitam um entendimento único. Da utilização de OCL, o que é relevante é que para as operações do sistema, se registe a sua definição em termos de pré e pós-condições, bem como em função dos resultados construídos. A construção dos resultados constitui um aspecto fundamental da análise do comportamento do sistema, visto ser possível escrever com rigor como é que as variáveis são construídas e verificar quais as alterações em relação ao seu anterior valor (acessível através de `pre`). Esta informação cumpre o seu propósito na concretização de como é que são modificados os objectos do sistema e quais são as condições que devem ser respeitadas. Ao serem escritas numa linguagem como a OCL, promove-se a inexistência de ambiguidade na interpretação das expressões. Ao mesmo tempo, o levantamento dessas expressões identifica elementos que são passíveis de serem utilizados noutros modelos. As expressões em OCL que especifiquem as condições em que é possível efectuar determinada actividade devem ser integradas nos diagramas comportamentais, sejam estes de estado ou de sequência.

A utilização de OCL apresenta uma outra vantagem quando aplicada numa fase de captura de requisitos. É que sendo a linguagem declarativa, permite ao utilizador descrever qual é a forma do resultado, que é manifestamente o que ele sabe, não tendo que se pronunciar sobre como construir o sistema para obter o resultado. A equipa de projecto pode capturar as condições para as quais o sistema deve funcionar, antes de saber qual é a constituição e arquitectura dos artefactos que darão origem à solução. A introdução de uma linguagem declarativa nesta etapa da análise possibilita que se adopte uma posição mais orientada à especificação do que é pretendido, como é externamente percebido, do que à elaboração de uma solução arquitectural que permita construir uma solução.

A solução arquitectural vai ser obtida por composição de todos os contributos obtidos na elaboração dos diagramas comportamentais para os casos de uso. O resultado dessa composição pode não ser a melhor solução em termos de padrões arquitecturais, mas é possível na fase de concepção aplicar técnicas de reengenharia e construção orientada por modelos, com o intuito de melhorar a solução, tendo em conta que as restrições identificadas devem continuar a ser válidas. No entanto a informação recolhida de forma rigorosa durante a especificação e registada sob a forma de expressões OCL não perde eficácia apesar da alteração do modelo arquitectural, porque sendo as expressões registadas durante a fase de análise incidem sobre objectos do mundo do negócio e estes não são alteráveis por influência de decisões arquitecturais.

6.6 Abordagens Alternativas

Na secção 3.8 foram referidas diversas abordagens alternativas, na comunidade académica, à proposta apresentada neste trabalho. Nessa secção a óptica pela qual se analisaram as diferentes propostas incidiu sobre os esforços que têm sido desenvolvidos para suprir as lacunas que a linguagem UML e os processos de desenvolvimento que a utilizam actualmente apresentam. Com esse intuito, apresentaram-se diversas abordagens, diferentes na forma e na estratégia, que procuram acrescentar à UML um maior poder expressivo que se julga necessário para efectuar de forma correcta as tarefas de análise de requisitos.

Após ter sido apresentado o processo proposto, com a integração de expressões rigorosas numa abordagem multi-vista da captura de requisitos, é importante referir outros trabalhos, ou linhas de investigação, que tem objectivos similares, ou que abordam a mesma problemática. Faz sentido apresentá-los nesta fase, de modo a que seja mais natural a comparação das abordagens e se possa ter uma percepção das diferentes

linhas de acção.

O trabalho de Mencl [Mencl 04] aborda a problemática do relacionamento entre diferentes casos de uso que são utilizados para especificar o comportamento de um sistema software. Neste trabalho começa-se por analisar e definir formalmente os aspectos relativos à construção de um sistema como sendo composto por um conjunto de casos de uso que podem ser compostos de forma a constituir comportamentos mais complexos. A motivação surge da necessidade de especificar qual é o comportamento de um sistema a partir de um conjunto de casos de uso representados de forma independente. Mencl labora em alternativas que permitam descrever os requisitos do sistema a construir, com a adição de informação que mostre como é que esses requisitos são concretizados pelos casos de uso.

Um modelo formal é construído de forma a capturar os conceitos de construção e composição de casos de uso, permitindo a sua utilização como ferramenta de análise da consistência das especificações de comportamento. O trabalho de Mencl representa a assemblagem e composição de casos de uso com o recurso a uma especialização das Protocol State Machines, que são uma variante das máquinas de estado, e utiliza esta definição para descrever o comportamento das tarefas que englobam a invocação de vários casos de uso. A camada de raciocínio proposta permite especificar o comportamento com recurso às novas máquinas de estado e foi utilizada para automatizar a conversão de casos de uso escritos textualmente para estes novos modelos. A abordagem considerada apresenta modificações ao modelo dos casos de uso, criando a noção de *Pro-cases*, que correspondem a novos mecanismos de especificação dos requisitos em que estão bem definidas as noções de composição e consistência interna e uma ferramenta foi construída para transformar casos de uso, escritos em linguagem natural, para Pro-cases. Posteriormente o raciocínio elaborado ao nível do refinamento das máquinas de estado é feito com base nestas novas descrições dos requisitos.

Ainda relacionado com a temática da comunicação entre o cliente e a equipa de projecto, o trabalho de [Johannisson 05] procura reduzir a distância conceptual entre as especificações formais em OCL e a linguagem que é utilizada para informalmente descrever os requisitos. O trabalho em questão parte do princípio que as especificações formais são necessárias para, através de métodos formais, se verificar a correcção do sistema que se está a construir. No entanto, como mecanismo de incorporação do cliente no processo de descrição dos requisitos e consequente conversão para notação rigorosa em OCL, foi criada uma ferramenta que permite exportar para linguagem natural o que está formalmente especificado. A premissa base, na qual o trabalho referido se sustenta, é a de que é necessário envolver no mesmo esforço de formalização pessoas

com diferentes graus de formação no uso de métodos formais, de modo a que se torne viável a gestão e manutenção dos requisitos expressos dessa forma. Como solução para o problema identificado, tal trabalho de incidiu na criação de uma ferramenta que traduz as especificações escritas em OCL para linguagem natural, de modo a que seja possível a quem não está familiarizado com a notação formal acompanhar e validar o que está a ser rigorosamente capturado. Um editor multi-língua é disponibilizado, podendo-se editar as especificações em OCL ou em língua natural que estão a ser construídas.

Este trabalho está integrado no sistema KeY [Beckert 07], cujo objectivo é a integração de técnicas formais de especificação no processo de engenharia de software. Em KeY podem escrever-se especificações de programas Java Card (que é um sub-conjunto de Java) que são concretizadas através de restrições expressas em OCL. O sistema transforma as expressões OCL e os programas Java Card, em provas que podem ser verificadas através de um provador de teoremas. Apesar de o trabalho de Johannison e a motivação que preside à formalização dos modelos UML estar de acordo com a proposta defendida nesta dissertação, este não é orientado primariamente à captura de requisitos através da identificação da informação fornecida via modelo dos casos de uso. A formalização faz-se principalmente ao nível dos diagramas estruturais e na especificação das operações de cada uma das classes. A transformação das expressões que regulam as associações entre entidades e as pré e pós-condições e invariante, em linguagem natural produz informação importante, mas seria interessante que essa transformação se concretizasse também ao nível da captura e descrição dos requisitos.

Uma outra abordagem que deve ser referida é a do Executable UML [Mellor 02], que se constitui como uma concretização de uma plataforma independente do modelo. Uma especificação em Executable UML é constituída por um conjunto de modelos, representados como diagramas, que originam um único documento com várias vistas. Essas vistas possibilitam que se especifiquem os dados, as entidades, através dos diagramas de classe, os objectos e o seu controlo interno, através do recurso aos diagramas de estado e por último, as acções que são descritas numa linguagem de acções (Action Semantics).

Esta concepção da abordagem permite que se formalize o conhecimento que se tem do sistema a construir, a um nível de abstracção elevado, através do recurso a modelos comportamentais e estruturais e com o recurso a uma linguagem que permite expressar que acções é que devem ser tomadas em cada momento da vida das entidades. É também possível conjugar a linguagem de restrições OCL com a linguagem de acções. A abordagem do Executable UML, tal como a desta dissertação, recorre às máquinas de estado para especificar o comportamento, embora a utilização de uma linguagem de

acções que poderia ser de mais alto nível, possa comprometer a esperada capacidade de independência dos modelos.

Um modelo em Executable UML especifica de forma completa a semântica do comportamento de determinada entidade, o que implica que é de facto um programa que a descreve. Sob esse prisma a independência da plataforma fica comprometida, visto que o detalhe a que a discriminação das acções obriga, torna a descrição demasiado concreta. No entanto, existem aspectos que não são especificados e que fazem sentido considerar nos sistemas actuais e que se referem à especificação de componentes distribuídos, à existência de processos concorrentes e mesmo à definição de classes e objectos necessários à concretização do sistema. Apesar de alguns destes aspectos pertencerem ao domínio das decisões associadas à construção e implementação do sistema, torna-se (como é defendido nesta dissertação) importante considerar estes aspectos durante a especificação do comportamento. Apesar de um compilador do modelo poder construir o sistema correcto com as decisões arquitecturais necessárias para a função pretendida, do ponto de vista da validação dos requisitos e associações entre as entidades, faz sentido incluir esta informação.

Dado que o Executable UML coloca a ênfase na descrição do comportamento das entidades como processo de descrição dos requisitos, a descoberta das restrições assume papel importante, e elas são expressas formalmente de modo a que possam ser verificadas de forma automática. A informação sobre as restrições pode também ser utilizada para que os compiladores de modelos possam efectuar as operações básicas de acesso a dados e interface com repositórios de dados. A descrição do comportamento das entidades é feita com recurso aos diagramas de estado, embora a óptica não seja centrada nos casos de uso mas sim nas entidades (as classes). A estratégia defendida no Executable UML assenta numa definição de controlo particionado por várias classes, de forma a que exista uma máquina de estados por cada uma delas, optando por em situações de maior complexidade, factorizar e criar novas classes de modo a simplificar a especificação de comportamento.

Apesar de esta estratégia fazer sentido porque permite lidar com a complexidade, retira o foco da especificação de controlo do caso de uso, para passar a considerar apenas as entidades. Uma crítica que pode ser feita ao Executable UML reside na importância marginal que dedica aos casos de uso, considerando-os apenas um mecanismo para garantir abstracção na recolha de requisitos. Após ter sido construído o modelo que corresponde ao domínio da solução, a estratégia defendida é que os casos de uso deixam de ser necessários, o que parece limitar seriamente a capacidade de elaborar sobre a componente dinâmica do sistema. Em coerência com a abordagem preconizada, em

Executable UML, os casos de teste são delineados a partir da informação das entidades, logo são feitos com bases em casos de uso que se supõem que fazem sentido, tendo em conta o comportamento (os métodos) exibido pela entidade. Este processo corresponde a uma técnica de abstracção reversa, em que em vez de se partir dos casos de uso e encontrar os modelos que os materializam, faz-se o oposto e a partir dos modelos é que se tentam encontrar os cenários. Esta estratégia está em contradição com o que se defende nesta dissertação, onde se assume que o princípio da construção do modelo não são os diagramas estruturais, mas antes as vistas que capturam os requisitos dos utilizadores.

Existem várias ferramentas que utilizam as noções do Executable UML e que permitem a validação dos modelos e a condução de testes por parte dos utilizadores. Entre essas podem citar-se o iUML [Raistrick 04] da Kennedy Carter, que é um ambiente de modelação e simulação, com a capacidade de geração de código em C e que utiliza como linguagem de acções a ASL (Action Specification Language). Na especificação dos diagramas de estado as acções são associadas às actividades entry de cada estado. Entre outras ferramentas que se podem referir encontram-se o Conformiq Qtronic, com utilização da QML como linguagem de acções (www.conformiq.com) e o Cassandra (ver http://www.omg.org/mda/mda_files/KnowGravity2004.pdf), entre outros.

Um outro trabalho que deve ser referido é o método de desenvolvimento de sistemas software reactivos designado por Fondue [Sendall 00, Sendall 02, Strohmeier 04]. Este método baseia-se no Fusion [Coleman 93] e aproveita deste a definição do processo mas acrescenta-lhe a utilização da UML e a definição de operação (*operation schema*) que consiste na adição da especificação das suas pré e pós-condições em OCL. Operações em Fondue são de forma declarativa descritas como sendo efeitos que são provocados no diagrama de estados que regula o comportamento do sistema. Dessa forma é necessário que sejam descritas as condições em que a chamada da operação pode ocorrer e o que acontece após essa invocação. A mudança de estado que ocorre como resultado de uma operação é descrita em termos dos objectos, atributos e relações entre as classes, isto é, em função do diagrama estrutural do sistema. Fondue propõe um modelo de descrição das operações, onde descreve em maior detalhe os efeitos da operação. Nem todas as entradas deste modelo são parte da definição OCL, embora exista um esforço para que sempre que possível se utilizem expressões OCL.

Os casos de uso são vistos em Fondue como sendo descritos por um sub-conjunto dos diagramas de estado que são utilizados para descrever a sequenciação das invocações de operações. Nessa perspectiva em Fondue a especificação das tarefas, se vistas como uma série de casos de uso que são invocados para completar determinado objectivo, é

feita com recurso a uma descrição de comportamento assente em máquinas de estado.

Em relação ao trabalho que se apresenta nesta dissertação, Fondue partilha o recurso a diagramas de estado para modelar o comportamento de um caso de uso e a utilização de expressões OCL para descrever os efeitos da invocação de uma operação. Em Fondue o caso de uso é prematuramente materializado na noção de operação, sendo que por vezes dá origem a que o sistema possua um conjunto de funcionalidades externas que são derivadas mais por efeito da definição de comportamento dos artefactos especificados do que resultam de requisitos expressos pelo utilizador. A criação de um nível de protocolo de comunicação entre as entidades, onde a semântica das construções, foi definida possibilitará a criação de uma camada de prototipagem que de momento não possui. Note-se que Fondue acrescenta notações visuais próprias e construções que não são norma da UML, embora um analista habituado à notação UML e aos modelos orientados aos objectos, rápida e facilmente apreende os conceitos. Também se sugerem alterações e extensões à OCL de forma a ser mais intuitiva e facilitada a descrição das operações.

Merece ainda ser referida a abordagem da Java Modeling Language (JML) [Burdy 05] que é uma interface comportamental para Java na qual se podem escrever asserções e testar contratos. Esta abordagem é muito interessante quando se pretende decorar o código com expressões retiradas dos contratos, mas tem o inconveniente de ser demasiado ligada a uma concretização numa linguagem de programação. Numa perspectiva de geração automática de modelos, a JML pode constituir uma interessante mais-valia visto que é possível, nesse caso, transportar as expressões OCL para o código gerado, sem que haja necessidade de intervenção dos analistas.

6.7 Resumo

Neste capítulo mostrou-se a aplicação do processo de modelação de sistemas software a dois casos de estudo, possibilitando que se demonstrasse as qualidades da abordagem proposta. A descrição de casos de uso de um Sistema de Comércio Electrónico e do Sistema de Documentação da Universidade, permitiu explorar e validar os vectores principais do processo. Foi detalhado o processo que define como é que a captura de requisitos de um sistema software é conduzida e foram devidamente explicitados os contributos que esta fase deverá proporcionar às fases subsequentes do processo de software. Foi apresentada o modo como a plataforma de suporte à prototipagem pode ser utilizada para animar as especificações de comportamento, no intuito de validar operacionalmente se o que foi especificado é realmente o que se pretende.

O processo iterativo e de refinamento sucessivo, com ganhos de conhecimento a cada caso de uso que se detalha, é concretizado nos exemplos dos casos de estudo. A utilização dos diversos diagramas foi apresentada e ilustrou-se o processo pelo qual a informação que é adquirida neles se propaga para os outros elementos do modelo. Estes casos de estudo permitem que se valide da aplicabilidade do processo proposto e se elabore sobre as consequências que dele se retiram para a melhoria do processo de software.

Foram apresentados outros trabalhos que se relacionam directamente com este por terem os mesmos objectivos e partilharem alguns dos princípios e soluções. Para cada um deles descreveram-se as suas principais características e, quando tal se justificou, compararam-se abordagens, objectivos e processos.

Após apresentar a forma como é aplicado o modelo de processo que é proposto, é necessário reflectir sobre quais são as vantagens que uma equipa de projecto obtém da sua utilização. O modelo de processo apresentado é direccionado à fase da captura de requisitos sendo na sua génese uma proposta sistémica de introdução de refinamento sucessivo nas peças de modelação que vão sendo obtidas. O seu propósito inicial consiste em detalhar de forma mais aturada a fase inicial do projecto de software com o recurso às ferramentas que as notações e metodologias orientadas aos objectos fornecem. A premissa inicial do trabalho baseou-se na necessidade de colocar em prática um processo que rigidamente conduzisse à criação de conhecimento. O detalhar da narrativa textual dos casos de uso com a indicação dos contratos que se definem para cada caso de uso possibilita que a captura de requisitos produza documentos substancialmente melhorados. A posterior descrição do comportamento do caso de uso com o recurso aos diagramas de estado já constitui um incremento notório de detalhe no processo. A necessidade que a equipa tem de descrever quais são os seus fios de execução e descrever em que condições cada um dos possíveis caminhos é percorrido, permite levantar informação que é relevante para a modelação do sistema.

Parece inclusive ser mais natural este procedimento do que aquele que advoga logo a passagem para a descrição das interacções com o recurso ao diagrama de sequência. A descrição de comportamento no diagrama de estados é uma tarefa que, pela própria natureza do formalismo das máquinas de estado, não é trivial sobretudo se a equipa pretender saber exactamente o que se passa em cada estado e o que desencadeia cada uma das transições. Este processo levanta de forma natural a identificação das entidades envolvidas e a identificação da mensagem. A justificação para a ocorrência da mensagem deriva das necessidades de descrição existentes no diagrama de estados e não surge apenas como um item da descrição da interacção no diagrama de sequência. Este

último refina a informação recebida, como se constatou nos exemplos deste capítulo, com outra informação que derive da descrição orientada aos objectos.

Desta forma, parece natural designar este modelo de processo como sendo bastante coeso na descrição sistemática e exaustiva dos casos de uso. A sua utilização para todos os casos de uso documenta rigorosamente todos os seus comportamentos e todas as interacções deles decorrentes. A utilização de uma linguagem rigorosa como OCL obriga a que ao mesmo tempo que se elaboram os diagramas se especifique de forma declarativa o caso de uso, sendo que as expressões OCL anotam os diagramas produzidos e são importantes para que se consiga uma descrição o mais completa exaustiva possível. Quando se procede à escrita em OCL das operações (vistas como transacções) decorrentes da análise do caso de uso, existe realimentação para os diagramas entretanto produzidos, porque a informação recolhida pode ser utilizada como guarda das transições ou como descrição de alguma das actividades internas dos estados (seja como pré, pós-condição ou invariante). As descrições OCL podem ser também utilizadas como descrição dos comandos que parametrizam a descrição de comportamento de um objecto estado na arquitectura de suporte. Estes comandos são operações muito específicas e a tradução de OCL para o código destas operações é simples e pode ser automatizado.

Por último, o processo favorece a noção de iteração durante a fase de análise, pois enquanto se está a especificar um caso de uso e a construir os diagramas que o descrevem, as adendas e correcções que são encontradas num determinado local são facilmente reproduzidas nos restantes diagramas, favorecendo uma iteração natural. O passo da espiral é bastante apertado e as alterações são facilmente incorporadas. O processo é multi-vista, mas não existe entre as vistas uma separação acentuada que dificulte o transporte de informação de um diagrama para outro.

Quando a iteração surge como resultado da validação operacional, a unidade de análise é o estado em que determinada condição não foi cumprida, ou não existe. Ao ter como unidade de análise o estado e não o caso de uso total, o impacto das alterações é menor e surge de forma mais controlada. Fazer reflectir as mudanças no resto do diagrama ou no diagrama de sequência apresenta igualmente um impacto reduzido, visto que é simples determinar o impacto que uma alteração num dos diagramas tem nos outros, bastando para isso localizar onde são descritas as operações associadas à mudança de estado.

Capítulo 7

Conclusões

Neste capítulo, começa-se por relembrar o trabalho realizado e destacar as suas principais contribuições. De seguida são apresentados os vários passos que constituem o modelo de processo proposto e são apresentadas algumas conclusões e razões que justificaram cada um desses passos. No final descreve-se o trabalho futuro que se perspectiva poder ser feito a partir das contribuições deste trabalho.

7.1 Introdução

O desenvolvimento de sistemas software complexos é uma tarefa árdua que exige da parte da equipa de projecto a utilização das ferramentas correctas, para que o resultado obtido seja o esperado. A utilização de uma metodologia que acompanhe e guie o processo de desenvolvimento é de importância crucial. As fases de análise e concepção são determinantes para a qualidade do produto final, na medida em que claramente condicionam o desenvolvimento que é realizado. A importância da construção de modelos do sistema, antes do seu desenvolvimento efectivo, é um aspecto determinante para que a equipa de projecto possa pensar o problema e utilizar o modelo como elemento de prototipagem.

O desenvolvimento de sistemas software usando modelos descritos na linguagem UML é hoje, em inúmeros e diferentes contextos, uma prática de facto. Os Casos de Uso afirmaram-se como modelos de comportamento muito adequados para clientes e analistas, pois, de uma forma diagramática simples, estabelecem uma linguagem comummente aceite por ambas as categorias de intervenientes no processo. Porém, os casos de uso são uma notação informal, útil a um nível inicial de captura de requisitos,

orientados à compreensão do processo de negócio como um todo lógico, não devendo ser portanto considerados, só por si, uma mais valia que resolve todas as carências conhecidas dos processos de desenvolvimento.

Assim, os casos de uso devem ser usados a muito alto nível, mas tendo presente que o analista deve possuir duas perspectivas do sistema a ser desenvolvido: a perspectiva de compreensão do negócio e dos requisitos, com os clientes, e a perspectiva de que tem que, com a equipa de desenvolvimento, garantir que vai desenvolver o que foi claramente definido e contratualizado ao nível dos requisitos.

Este parece ser um dos pontos fundamentais da abordagem proposta, comparativamente com outras abordagens que procuram apenas formalizar o que, de momento, é um artefacto de enorme valia e utilização exactamente por ser informal e, portanto, de fácil acesso semântico por parte dos clientes.

Mas o correcto desenvolvimento necessita que se introduza mais rigor no processo, sendo por nós considerada um premissa fundamental que toda a introdução de rigor e formalismo no processo seja o mais possível abstraída da visão cliente, ainda que sirva para demonstrar ao cliente os erros de captura de requisitos.

O contributo principal deste trabalho consistiu na proposta de um modelo de processo de modelação que recorrendo à UML, como notação, apresenta a construção do modelo, mais concretamente da fase de análise. O processo de modelação assenta em três aspectos fundamentais:

1. a construção de uma abordagem unificada com recurso a várias vistas do modelo e que tem os casos de uso como ponto de partida da captura de requisitos;

2. a introdução no processo de modelação de rigor, através da inclusão de expressões em OCL que formalizam as condições em que os casos de uso podem ser invocados,

3. uma preocupação em operacionalizar as expresões de comportamento dos casos de uso através do recurso a prototipagem, como mecanismo de validação operacional.

7.2 Trabalho Realizado

Esta proposta apresentou uma alteração ao modelo de processo de modelação que é usualmente apresentado e parte do pressuposto que a fase de análise é essencial para o desenvolvimento do sistema. Essa alteração visa acrescentar rigor aos documentos e modelos que são criados durante a fase de análise.

De forma a enquadrar os resultados obtidos recorde-se a definição de Jayaratna [Jayaratna 94], feita no âmbito das metodologias de análise e concepção.

> *Metodologia* é uma forma explícita de estruturarmos as nossas acções e pensamento. As metodologias contêm modelos que reflectem perspectivas particulares da realidade baseadas num conjunto de paradigmas filosóficos. Uma metodologia deve dizer quais os passos a executar e como executar esses passos, mas, ainda mais importante, as razões pelas quais esses passos devem ser executados, naquela ordem particular.

Esta definição tem o cuidado de incluir a razão de ser da metodologia. A definição dá igualmente importância à justificação dos passos, e da sua ordem, pela importância que assumem no contexto da análise e concepção. Embora não se considere que se tenha desenvolvido uma metodologia, a nossa abordagem contém um conjunto de passos que se propõe que sejam dados pela equipa de projecto.

Assim, na nossa abordagem, muito cedo no processo se procura dar um tratamento rigoroso aos casos de uso, fundamentalmente através do facto de se propor que a equipa de projecto proceda de forma a que:

1. Faça a atomização dos vários passos (transacções) que constituem um caso de uso completo. Definiu-se a forma como um caso de uso pode ser separado modularmente em vários passos e possibilitou-se que cada um desses passos fosse analisado e descrito de forma autónoma. Esta modularidade na abordagem possibilita que o processo possa ser aplicado a sistemas complexos, cuja complexidade é intrínseca e não pode ser diminuída. Como a abordagem seguida divide os casos de uso numa sequência de passos mais simples, é possível aplicar o processo aos casos de uso dos sistemas complexos diminuindo a complexidade do tratamento e das peças envolvidas. Este mecanismo de decomposição dos casos de uso em elementos mais simples possibilita que sem diminuir a complexidade de algo que é inerentemente complexo se possa proceder, de igual forma, às tarefas de análise dos sistemas complexos e dos outros, mais simples;

2. Transforme os casos de uso, modularmente, em máquinas de estado, de semântica rigorosa, criando um nível de observação comportamental dos requisitos ainda numa fase muito pouco funcional (no sentido das operações a implementar no sistema software). A construção da máquina de estados do caso de uso foi definida como sendo o resultado da composição das máquinas de estado que se podem definir para cada passo do caso de uso;

3. Abstraindo a este nível de observação se as operações a que nos referimos são apenas de negócio ou se virão a ser computacionais, ou seja, abstraindo da natureza física final das entidades que designamos na modelação por Sistema ou Sub-sistema;

4. Introduzindo a este nível OCL como linguagem lógica para expressão de propriedades e de verificação da consistência de característica comportamentais. Com a utilização da OCL como linguagem rigorosa manteve-se o foco de atenção no mesmo meta-modelo, não se criando modelos em notações paralelas e ortogonais à UML;

5. Realizando iterações muito curtas sempre centradas na verificação dos casos de uso, quer tomando-os isolados, quer depois tomando-os de forma estrutural, ou seja, considerando as suas ligações a outros casos de uso (conforme $<< include >>$ ou $<< extend >>$, cujas semânticas são bem estabelecidas). O processo favorece a introdução de iteração durante a fase de análise, pois enquanto se está a especificar um caso de uso e a construir os diagramas que o descrevem, as adendas e correcções que são encontradas num determinado local são facilmente reproduzidas nos restantes diagramas, favorecendo uma iteração natural. O passo da espiral é bastante apertado e as alterações são naturalmente incorporadas. O processo é multi-vista, mas não contempla uma separação tal entre as vistas que dificulte o transporte de informação de um diagrama para outro;

6. Gradualmente, realizando refinamentos no nível de análise dos casos de uso, em especial ao realizar os seus mapeamentos noutros modelos/diagramas, procurando identificar, ainda com os clientes, que entidades são do mundo do modelo de domínio, as entidades de negócio, e que entidades vão ser do domínio da aplicação, ou seja, computacionais. Este refinamento estará correcto e coerente se apenas em determinadas fronteiras bem limitadas do sistema software se realizar a transformação de entidades, ou seja, as comunicações e interfaces entre um nível e o outro;

   No limite, o processo terminará com 3 tipos de modelos: apenas "business" (de negócio), apenas "computacionais" e de "fronteira". Neste momento, o processo de análise estará provavelmente completo e quer a posterior codificação quer a posterior manutenção do sistema deve ser muito mais clara em termos de localização de modificações. Esta é a vantagem de o processo ser inerentemente multi-vista, e

7. Animando as expressões de comportamento através de um mecanismo de suporte à prototipagem que permite estabelecer um realimentação mais rápida e eficaz, ainda ao nível da fase de análise. Quando a iteração que o processo promove surge como resultado da validação operacional, a unidade de análise é o estado em que determinada condição não foi cumprida, ou não existe. Ao ter como unidade de análise o estado e não o caso de uso total, o impacto das alterações é menor e surge de forma mais controlada.

Apresentadas que foram as contibuições do trabalho, importa situá-lo no ecosistema de esforços que pretendem dotar o processo de desenvolvimento de software de ferramentas, sejam metodologias, modelos ou processos, que reforcem a fase de análise e mais concretamente a captura de requisitos. Deste modo, nada impede que todo o posterior esforço de melhor compreender a própria análise versando uma mais correcta fase de desenvolvimento do sistema, não possa ser agora realizada recorrendo a muitos e meritórios trabalhos que foram sendo desenvolvidos, mas que se centram na utilização de casos de uso absolutamente computacionais como modelos orientadores da validade das implementações, fase que será, naturalmente, absolutamente necessária também. No nosso caso, todas as transformações e arquitectura de suporte que foram usadas na fase inicial do processo, podem de novo ser utilizadas, sendo certo que agora estaremos a analisar comportamento referente a operações da camada computacional. Porém, a metodologia é exactamente a mesma, os artefactos são os mesmos, mas todos são agora referenciados (*"tipados"*) como sendo apenas entidades e operações de implementação.

Assim, o processo apresentado pode mesmo ser considerado como possuindo a propriedade muito importante da *"ortogonalidade"*, ou seja, é aplicável em diferentes fases do projecto, dos requisitos à implementação, sem introduzir formalismos nos modelos informais que tão bons resultados têm obtido junto dos clientes.

O processo necessita de maior experimentação e aplicação prática não académica, para que se possa proceder a uma mais efectiva avaliação [dCS04].

7.3 Trabalho Futuro

Nesta secção abordam-se alguns aspectos que constituem linhas de investigação que se podem traçar na sequência desta dissertação. Perspectiva-se que haja ainda muito trabalho para fazer, devido à importância do tema e à necessidade que existe nos processos de desenvolvimento de encontrar respostas para as lacunas identificadas na fase de análise. Foi no entanto necessário estudar todas as componentes, vantagens e

limitações do mesmo para que, agora, no final deste trabalho, se tenham ideias muito mais claras sobre como e que partes do processo podem ser melhor trabalhadas visando a sua automatização parcial ou total.

7.3.1 Validação sintáctica e semântica dos modelos

O processo proposto para a construção do modelo, principalmente no que respeita à fase de análise, faz com que para cada caso de uso se construa a sua especificação e depois se inicie um processo de descrição de outros diagramas que detalham o seu comportamento e conduza um processo de enriquecimento do modelo. O modelo final é a junção de todos estes pedaços de diagramas, sendo que é necessário validar o modelo obtido para aquilatar se sintáctica e semanticamente é coerente. A verificação do modelo obtido é uma área que pode ser iniciada no seguimento deste trabalho, na medida em que o modelo obtido deve ser coerente e completo em todas as suas vertentes. O próprio processo só deverá estar terminado quando o modelo for totalmente válido.

Uma outra linha de trabalho que se pode traçar a partir desta dissertação tem a ver com a validação que pode ser feita do modelo em termos da sua semântica. Os modelos obtidos dizem muito sobre o sistema que modelam (pelo menos essa é a mensagem deste trabalho), pelo que é possível inferir a complexidade da aplicação a partir do modelo produzido. Ao mesmo tempo também se poderá detectar falhas de modelação sempre que os modelos produzidos não correspondam ao sistema a desenvolver.

7.3.2 Construção baseada em casos de uso

O processo proposto detalha os casos de uso como sendo um conjunto de passos, que possuem eles próprios uma descrição do seu comportamento. A obtenção do diagrama de comportamento para um caso de uso faz-se por composição das diversas expressões de comportamento dos passos do caso de uso. O conceito pode ser especializado para as situações em que se pretender compor casos de uso, situação em que deveria ser possível de forma automática criar as composições dos diagramas de estados e sequência associados. Estes aspecto é especialmente relevante quando se pretender reorganizar as tarefas de um sistema e se pretender rearranjar comportamentos definidos pelos casos de uso.

7.3.3 Coesão entre vistas

O processo proposto ao efectuar a especificação dos casos de uso e correspondente construção dos diagramas comportamentais, identifica e representa entidades que são de natureza distinta. Dessas entidades umas são mais do domínio do problema, isto é são do mundo do negócio, e outras que pertencem à fronteira com o modelo da aplicação, ou são mesmo entidades da aplicação. Essas entidades estão conceptualmente em "vistas" distintas do modelo e importa saber qual é a relação entre elas e o impacto que alterações provocam a essas vistas. Uma linha de trabalho futuro corresponde a estudar e melhorar a correspondência entre as vistas que o processo permite criar, isto é, entre a vista do modelo de negócio e a vista do modelo da aplicação. Pretende-se, desta forma, tornar o processo mais orientado aos modelos (*model driven*), no sentido da introdução de uma maior automatização do mesmo e estudar o processo sistemático que permita passar da *"business user interface"* para a *"user interface"* e desta para a *"graphical user interface"*.

7.3.4 Ambiente visual de prototipagem

Uma vez que a modelação em UML assenta na mais-valia dos diagramas serem um instrumento gráfico, facilmente manipulável e perceptível tanto para os engenheiros como para os clientes (pelo menos alguns desses diagramas), é importante que as melhorias que se introduzam no processo de modelação possam ser também complementadas com a capacidade de visualização. A utilização de informação não visual nos modelos, embora muito importante, é frequentemente desvalorizada por não ser entendida tão facilmente e não se perceber imediatamente o seu papel. A utilização de expressões que descrevem restrições ao comportamento de entidades e às associações existentes entre elas, pode alimentar a camada de interface, e a validação operacional dos modelos pode ser visualizada graficamente.

A criação de um componente visual que simule e anime os cenários dos casos de uso, permitiria, de forma simples e integrada, a validação dos requisitos com a mais-valia que seria dada pela interactividade que o ambiente visual proporciona. O cliente também se poderia aperceber visualmente dos diversos fios de execução distintos e das interligações existentes entre diferentes casos de uso.

Anexo OCL

TIPOS PRIMITIVOS

Tipo	**Descrição**	**Valores**	**Operações**
`Boolean`	Um valor lógico	*true, false*	$=$, $<>$, *and*, *xor*, *not*, *implies*, $if\_then\_else\_endif$
`Integer`	Um número inteiro de qualquer tamanho	$-1, 0, 1, ...$	$=$, $<>$, $>$, $<$, $>=$, $<=$, $*$, $+$, $-$ (unário), $-$ (binário), / (real), $abs()$, $max(b)$, $min(b)$, $mod(b)$, $div(b)$
`Real`	Um número real de qualquer tamanho	$1.5, ...$	$=$, $<>$, $>$, $<$, $>=$, $<=$, $*$, $+$, $-$ (unário), $-$ (binário), /, $abs()$, $max(b)$, $min(b)$, $round()$, $floor()$
`String`	Uma sequência de caracteres	'a', 'Maria'	$=$, $<>$, $size()$, $concat(s2)$, $substring(from, to)$ $(1 <= from <= to <= size)$, $toReal()$, $toInteger()$

Nota(1): As operações indicadas com parentesis são aplicadas com ".", mas os parentesis podem ser omitidos.

Nota(2): Exemplo $if\_then\_else\_endif$: $title = (if$ isMale $then$ 'Mr.' $else$ 'Ms.' $endif)$

Tabela 7.1: Tipos primitivos

COLECÇÕES E TUPLOS

Descrição	**Sintaxe**	**Exemplos**
Colecção abstracta de elementos do tipo T	`Collection(T)`	
Colecção não ordenada, sem duplicados	`Set(T)`	$Set\{1, 2\}$
Colecção ordenada, que permite duplicados	`Sequence(T)`	$Sequence\{1, 2, 1\}$ $Sequence\{1..4\}$ (o mesmo que $Sequence\{1, 2, 3, 4\}$)
Colecção ordenada, sem duplicados	`OrderedSet(T)`	$OrderedSet\{2, 1\}$
Colecção não ordendada, que permite duplicados	`Bag(T)`	$Bag\{1, 1, 2\}$
Tuplo (com identificação das partes constituintes)	`Tuple(field1: T1, ..., fieldN: Tn)`	$Tuple\{idade : Integer = 5, nome : String =' Maria'\}$ $Tuple\{nome =' Maria', idade = 5\}$

Nota(1): São tipos de valores: "="e "<>"comparam valores e não referências

Nota(2): Os componentes de um tuplo podem ser acedidos com "."como em "$t1.nome$"

Tabela 7.2: Colecções e Tuplos

OPERAÇÕES SOBRE `Collection(T)`

Operação	**Descrição**
`size(): Integer`	O número de elementos nesta colecção ($self$)
`isEmpty(): Boolean`	$size = 0$
`notEmpty(): Boolean`	$size > 0$
`includes(object: T): Boolean`	$True$ se *object* é um elemento de $self$
`excludes(object: T): Boolean`	$True$ se *object* não é um elemento de $self$
`count(object: T): Integer`	O número de ocorrências de *object* em $self$
`includesAll(c2: Collection(T)): Boolean`	$True$ se $self$ contém todos os elementos de $c2$
`excludesAll(c2: Collection(T)): Boolean`	$True$ se $self$ não contém nenhum dos elementos de $c2$
`sum(): T`	A soma de todos os elementos de $self$ (T tem de suportar "+")
`product(c2: Collection(T2)): Set(Tuple(first:T, second:T2))`	O produto cartesiano de $self$ com $c2$

Nota(1): As operações sobre as colecções são aplicadas com "$->$"e não com "."

Tabela 7.3: Operações sobre `Collection(T)`

EXPRESSÕES COM `Iterator` SOBRE `Collection(T)`

Expressão Iterator	Descrição
`iterate(iterator: T; accum: T2 = init | body)`	Devolve o valor final de um acumulador que, após inicialização, é actualizado com o valor de *body* aplicado a cada elemento da colecção $fonte$.
`exists(iterators | body) : Boolean`	$Verdadeiro$ se *body* for verdadeiro para pelo menos um dos elementos da colecção $fonte$. Permite múltiplas variáveis $Iterator$.
`forAll(iterators | body): Boolean`	$Verdadeiro$ se *body* é verdadeiro para todos os elementos da colecção $fonte$. Permite múltiplas variáveis $Iterator$.
`one(iterator| body): Boolean`	Devolve *verdadeiro* se existe exactamente um elemento da colecção $fonte$ para o qual *body* é verdadeiro.
`isUnique(iterator| body): Boolean`	Devolve *verdadeiro* se *body* resulta num valor diferente para cada elemento da colecção $fonte$.
`any(iterator | body): T`	Devolve qualquer elemento da colecção $fonte$ para o qual *body* é verdadeiro. Devolve *null* se não existir nenhum.
`collect(iterator| body): Collection(T2)`	Colecção dos elementos resultantes de aplicar *body* a cada elemento do conjunto $fonte$. O resultado é aplanado.
`select(iterator | body): Collection(T)`	Colecção dos elementos da colecção $fonte$ para os quais *body* é verdadeiro. O resultado é uma colecção do mesmo tipo que a colecção $fonte$.
`reject(iterator | body): Collection(T)`	Colecção dos elementos da colecção $fonte$ para os quais *body* é falso. O resultado é uma colecção do mesmo tipo que a colecção $fonte$.
`collectNested(iterator | body): CollectionWithDuplicates(T2)`	Colecção dos elementos (admitindo duplicados) que resulta de aplicar *body* (do tipo $T2$) a todos os elementos da colecção $fonte$. O resultado não é aplanado. Conversão entre colecções: $Set->Bag, OrderedSet->Sequence$
`sortedBy(iterator | body): OrderedCollection(T)`	Devolve uma colecção ordenada de todos os elementos da colecção $fonte$ por ordem crescente do valor da expressão *body*. O tipo $T2$ tem de suportar o operador $<$. Conversão entre colecções: $Set->OrderedSet, Bag->Sequence$

Nota(1): A declaração da variável $Iterator$ pode ser omitida se não existir ambiguidade

Tabela 7.4: Expressões com `Iterator` sobre `Collection(T)`

OPERAÇÕES SOBRE `Set(T)`

Operação	**Descrição**
`=(s: Set(T)) : Boolean`	$self$ e s possuem os mesmos elementos?
`union(s: Set(T)): Set(T)`	A união de $self$ com o conjunto s.
`union(b: Bag(T)): Bag(T)`	A união de $self$ com o Bag b.
`intersection(s: Set(T)): Set(T)`	Intersecção de $self$ com s.
`intersection(b: Bag(T)): Set(T)`	Intersecção de $self$ com b.
`-(s: Set(T)) : Set(T)`	Os elementos de $self$ que não estão em s.
`including(object: T): Set(T)`	Conjunto contendo todos os elementos de $self$ mais o $object$.
`excluding(object: T): Set(T)`	Conjunto contendo todos os elementos de $self$ menos o $object$.
`symmetricDifference(s: Set(T)): Set(T)`	Conjunto de todos os elementos que ou estão em $self$ ou em s, mas não em ambos.
`flatten() : Set(T2)`	Se T é uma colecção, o resultado é o conjunto de todos os elementos de todos os elementos de $self$. Senão é o resultado é $self$.
`asOrderedSet(): OrderedSet(T)`	$OrderedSet$ com todos os elementos de $self$ numa ordem indefinida.
`asSequence(): Sequence(T)`	$Sequence$ com todos os elementos de $self$ numa ordem indefinida.
`asBag(): Bag(T)`	Um Bag com todos os elementos de $self$.

Tabela 7.5: Operações sobre `Set(T)`

OPERAÇÕES SOBRE `Bag(T)`

Operação	**Descrição**
`=(bag: Bag(T)) : Boolean`	*Verdadeiro* se *self* e *bag* contêm os mesmos elementos o mesmo número de vezes.
`union(bag: Bag(T)): Bag(T)`	A união de *self* e *bag*.
`union(set: Set(T)): Bag(T)`	A união de *self* e *set*.
`intersection(bag:Bag(T)): Bag(T)`	A intersecção de *self* e *bag*.
`intersection(set: Set(T)): Set(T)`	A intersecção de *self* e *set*.
`including(object: T): Bag(T)`	*Bag* com todos os elementos de *self* e mais *object*.
`excluding(object: T): Bag(T)`	*Bag* com todos os elementos de *self* excepto *object*.
`flatten() : Bag(T2)`	Se T é do tipo colecção, devolve um *Bag* com todos os elementos de todos os elementos de *self*; Senão devolve *self*.
`asSequence(): Sequence(T)`	*Sequence* com todos os elementos de *self* numa ordem indefinida.
`asSet(): Set(T)`	*Set* com todos os elementos de *self* sem duplicados.
`asOrderedSet(): OrderedSet(T)`	*OrderedSet* com todos os elementos de *self* numa ordem indefinida e sem duplicados.

Tabela 7.6: Operações sobre `Bag(T)`

OPERAÇÕES SOBRE `Sequence(T)`

Operação	Descrição
`=(s: Sequence(T)) : Boolean`	*Verdadeiro* se $self$ contem todos os elementos de s na mesma ordem.
`union(s: Sequence(T)): Sequence(T)`	*Sequence* com todos os elementos de $self$ seguidos de todos os elementos de s.
`flatten() : Sequence(T2)`	Se T é uma colecção, o resultado é um conjunto com todos os elementos de todos os elementos de $self$. Senão o resultado é $self$.
`append(object: T): Sequence(T)`	*Sequence* com todos os elementos de $self$ seguidos de *object*.
`prepend(obj: T): Sequence(T)`	*Sequence* com *object* seguido de todos os elementos de $self$.
`insertAt(index : Integer, object : T) : Sequence(T)`	*Sequence* que consiste de $self$ com *object* inserido na posição *index*, com $1 <= index <= size + 1$
`subSequence(from : Integer, to : Integer) : Sequence(T)`	*Sequence* contendo os elementos de $self$ desde a posição $from$ até à posição to, inclusive, sendo $1 <= from <= to <= size$
`at(i : Integer) : T`	O elemento da posição i de $self$, com $1 <= i <= size$
`indexOf(object : T) : Integer`	O índice da posição de *object* em $self$.
`first() : T`	O primeiro elemento de $self$.
`last() : T`	O último elemento de $self$.
`including(object: T): Sequence(T)`	*Sequence* incluindo todos os elementos de $self$ e ainda *object* adicionado no fim.
`excluding(object: T): Sequence(T)`	*Sequence* com todos os elementos de $self$ excluindo todas as ocorrências de *object*.
`asBag(): Bag(T)`	*Bag* contendo tos os elementos de $self$ incluindo duplicados.
`asSet(): Set(T)`	*Set* contendo todos os elementos de $self$ removendo duplicados.
`asOrderedSet(): OrderedSet(T)`	Um *OrderedSet* contendo todos os elementos de $self$, na mesma ordem, removendo os duplicados.

Tabela 7.7: Operações sobre `Sequence(T)`

OPERAÇÕES SOBRE `OrderedSet(T)`

Operação	**Descrição**
`append(object: T): OrderedSet(T)`	Conjunto ordenado de todos os elementos de *self* seguidos de *object*.
`prepend(object: T): OrderedSet(T)`	*OrderedSet* com *object* seguido de todos os elementos de *self*.
`insertAt(index : Integer, object : T) : OrderedSet(T)`	*OrderedSet* que consiste em *self* com *object* inserido na posição *index*.
`subOrderedSet(lower : Integer, upper : Integer) : OrderedSet(T)`	O subconjunto ordenado de *self* com os elementos desde a posição *lower* até à posição *upper*, inclusive, com $1 <= lower <= upper <= size$
`at(i : Integer) : T`	O elemento na posição i de *self*, com $1 <= i <= size$
`indexOf(object : T) : Integer`	A posição de *object* nos elementos de *self*.
`first() : T`	O primeiro elemento de *self*.
`last() : T`	O último elemento de *self*.

Tabela 7.8: Operações sobre `OrderedSet(T)`

TIPOS ESPECIAIS

Tipo	**Descrição**
`OclAny`	Supertipo de todos os tipos excepto as colecções. Todas as classes num modelo UML herdam todas as operações definidas para $OclAny$.
`OclVoid`	O tipo $OclVoid$ é um tipo que está conforme com todos os outros tipos. Tem apenas uma única instância chamada *null*.
`OclInvalid`	O tipo $OclInvalid$ é um tipo que está conforme com todos os outros tipos. Tem apenas uma única instância chamada *invalid*.
`OclMessage`	Tipo *template* com parâmetro T para ser substituído por uma operação concreta ou por um tipo de sinal. Usado nalgumas pós-condições que precisam restringir as mensagens enviadas durante a execução da operação.
`OclType`	

Tabela 7.9: Tipos Especiais

OPERAÇÕES DEFINIDAS EM `OclAny`

Tipo	**Descrição**
`=(object2 : OclAny) : Boolean`	*Verdadeiro* se *self* é o mesmo objecto que *object*2
`<>(object2 : OclAny) : Boolean`	*Verdadeiro* se *self* é um objecto distinto de *object*2
`oclIsNew() : Boolean`	Só pode ser usado como pós-condição. *Verdadeiro* se *self* foi criado durante a execução da operação.
`oclAsType(t : OclType) : T`	Operação de *Cast* (conversão de tipos). Útil para conversão mais específica.
`oclIsTypeOf(t: OclType) : Boolean`	*Verdadeiro* se *self* é do tipo t
`oclIsKindOf(t : OclType) : Boolean`	*Verdadeiro* se *self* é do tipo t ou um subtipo de t
`oclIsInState(s : OclState) : Boolean`	*Verdadeiro* se *self* está no estado s.
`oclIsUndefined() : Boolean`	*Verdadeiro* se *self* é igual a *null* ou a *invalid*.
`oclIsInvalid() : Boolean`	*Verdadeiro* se *self* é igual a *invalid*.
`allInstances() : Set(T)`	Operação estática que devolve todas as instâncias dum classificador.

Tabela 7.10: Operações definidas em `OclAny`

PROPRIEDADES DEFINIDAS EM `OclMessage`

Operação	**Descrição**
`hasReturned() : Boolean`	Devolve *Verdadeiro* se o tipo de *template* do parâmetro for uma invocação, e a operação invocada tiver retornado um valor.
`result()`	Devolve o resultado da operação invocada, se o tipo de *template* do parâmetro for uma invocação, e a operação invocada tiver retornado um valor.
`isSignalSent() : Boolean`	Devolve *Verdadeiro* se a *OclMessage* representa o envio de um sinal UML.
`isOperationCall() : Boolean`	Devolve *Verdadeiro* se a *OclMessage* representa o envio de uma chamada a uma operação UML.
`parameterName`	O valor do parâmetro mensagem.

Tabela 7.11: Propriedades definidas em `OclMessage`

Bibliografia

[Amyot 00] Daniel Amyot e Gunter Mussbacher. On the extension of UML with use case maps concepts. Em Andy Evans, Stuart Kent, e Bran Selic, editores, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939, páginas 16–31. Springer, 2000.

[Andrews 91] Gregory R. Andrews. *Concurrent Programming - principles and practice.* The Benjamin Cummings Publishing Company, 1991.

[Armour 01] F. Armour e G. Miller. *Advanced Use Case Modeling.* Addison-Wesley, 2001.

[Bauer 01] Bernhard Bauer, Jorg P. Muller, e James Odell. *Agent-Oriented Software Engineering*, capítulo Agent UML: A Formalism for Specifying Multiagent Interaction, páginas 91–103. Springer-Verlag, 2001.

[Beckert 07] Bernhard Beckert, Reiner Hähnle, e Peter H. Schmitt, editores. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[Bernardez 04] Beatriz Bernardez, Amadore Duran, e Marcela Genero. Empirical evaluation and review of a metrics-based approach for use case verification. *Journal of Research and Practice in Information Technology*, 36(4), 2004.

[Boehm 88] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[Booch 94] Grady Booch. *Object Oriented Analysis and Design with Applications.* Benjamin Cummigs, 2ª edição, 1994.

[Booch 98] Grady Booch, James Rumbaugh, e Ivar Jacobson. *The Unified Modeling Language - User Guide*. Addison-Wesley, 1998.

[Booch 04] Grady Booch. An MDA Manifesto. Em D. Frankel e J. Parodi, editores, *The MDA Journal*. Meghan-Kiffer Press, 2004.

[Booch 05] Grady Booch, James Rumbaugh, e Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 2ª edição, 2005.

[Briand 02] L. Briand e Y. Labiche. A uml-based approach to system testing. *J. Software and Systems Modeling*, páginas 10–42, 2002.

[Burdy 05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, e Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, Junho 2005.

[Campos 06] José Creissac Campos e António Nestor Ribeiro. Uml no desenvolvimento de sistemas interactivos. Em *Actas da 2a. Conferência Nacional em Interacção Pessoa-Máquina*, Braga, Portugal, 2006.

[Chrissis 06] Mary Beth Chrissis, Mike Konrad, e Sandy Shrum. *CMMI: Guidelines for Process Integration and Product Improvement*. Addison Wesley, 2006.

[Chung 99] Lawrence Chung, Brian A. Nixon, Eric Yu, e John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, October 1999.

[Clark 00] Tony Clark, Andy Evans, Stuart Kent, Steve Brodsky, e Steve Cook. A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta Modeling Approach. Relatório técnico, Precise UML Group and IBM, 2000.

[Clark 01] Tony Clark, Andy Evans, e Stuart Kent. Engineering modelling languages: A precise meta-modelling approach. Relatório técnico, 2001.

[Coad 91] Peter Coad e Edward Yourdon. *Object-Oriented Analysis*. Prentice-Hall, 1991.

[Cockburn 01] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[Coleman 93] Derek Coleman. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1993.

[Consortium 02] 2U Consortium. Unanbiguous uml (2u) 2nd revised submission to uml 2 infrastructure rfp. Relatório técnico, 2002.

[Constantine 01] L. Constantine e L. Lockwood. *Object Modeling and User Interface Design*, capítulo Structure and Style in Use Cases for User Interface Design, páginas 245–279. Object Technology. Addison-Wesley, 2001.

[dCS04] Alberto António de Chalupa Sampaio. *Comparação e Classificação de Métodos de Avaliação do Processo do Software Utilizando uma Metodologia Numérica e Exploratória*. Tese de Doutoramento, Escola de Engenharia, Universidade do Minho, 2004.

[DeMarco 79] Tom DeMarco. *Structural Analysis and System Specification*. Prentice-Hall, 1979.

[Dix 04] Alan Dix, Janet Finlay, Gregory Abowd, e Russell Beale. *Human-Computer Interaction*, capítulo 15. Prentice-Hall, 2004.

[Douglass 98] Bruce Powel Douglass. *Real Time UML - Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998.

[Douglass 04] Bruce Powell Douglass. *Real Time UML: Advances in the UML for Real-Time Systems*. Addison-Wesley, 2004.

[dS00] Paulo Pinheiro da Silva e Norman W. Paton. UMLi: The unified modeling language for interactive applications. Em Andy Evans, Stuart Kent, e Bran Selic, editores, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939, páginas 117–132. Springer, 2000.

[dS02] Paulo Pinheiro da Silva. *Object Modelling of Interactive Systems: The UMLi approach*. Tese de Doutoramento, Department of Computer Science - University of Manchester, 2002.

[d'Souza 99] Desmond d'Souza e Alan Wills. *Objects, Components, and Frameworks with UML.* Addison-Wesley, 1999.

[Duran 04] A. Duran, B. Bernardéz, M. Genero, e M. Piattini. Empirically driven use case metamodel evolution. Em "Thomas Baar, Alfred Strohmeier, Ana Moreira, e Stephen J. Mellor", editores, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, volume 3273 de *LNCS*, páginas 1–11. Springer, 2004.

[Engels 00] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, e Stefan Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in uml. Em *UML 2000 - The Unified Modeling Language. Advancing the Standard: Third International Conference, York, UK, October 2000*, Lecture Notes in Computer Science. Springer-Verlag, 2000.

[Eshuis 02] H. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling.* Tese de Doutoramento, University of Twente, Enschede, The Netherlands, 2002.

[Evans 98] A. Evans, R. France, K. Lano, e B. Rumpe. The uml as a formal modelling notation. Em Springer Verlag, editor, *UML'98 - Beyond the notation*, 1998.

[Evans 00] Andy Evans, Stuart Kent, e Bran Selic, editores. *UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings*, volume 1939 de *Lecture Notes in Computer Science*. Springer, 2000.

[Fernandes 00] João Miguel Fernandes. *MIDAS: Metodologia Orientada aos Objectos para Desenvolvimento de Sistemas Embebidos.* Tese de Doutoramento, Escola de Engenharia, Universidade do Minho, Fevereiro 2000.

[Filman 04] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, e Mehmet Aksit. *Aspect-Oriented Software Development.* Addison Wesley, 2004.

[Fowler 97] Martin Fowler. *UML Distilled.* Addison-Wesley, 1997.

[Fowler 04] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language - UML 2.0.* Addison-Wesley, 3ª edição, 2004.

[France 00] Robert France, Emanuel Grant, e Jean-Michel Bruel. Umltranz: An uml-based rigorous requirements modeling technique. Relatório técnico, Colorado State University, 2000.

[Frolhich 00] P. Frolhich e J. Link. Automated test case generation from dynamic models. Em *14th European Conference Object-Oriented Programming*, 2000.

[Gamma 94] Erich Gamma, Richard Helm, Ralph Johnson, e John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[Gomaa 00] Hassan Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML.* Addison-Wesley, 2000.

[Goñi 04] Agustín Goñi e Yadran Eterovic. Building precise uml constructs to model concurrency using ocl. Em "Thomas Baar, Alfred Strohmeier, Ana Moreira, e Stephen J. Mellor", editores, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, páginas 212–225. Springer-Verlag, 2004.

[Grieskamp 00] Wolfgang Grieskamp e Markus Lepper. Using use cases in executable z. Em *ICFEM*, páginas 111–120, 2000.

[Group 00] The VDM Tool Group. The rose-VDM++ link. Relatório técnico, IFAD, Outubro 2000.

[Group 04] Object Management Group. UML 2.0 Superstructure, Available Specification, document ptc/04-10-02. Relatório técnico, Object Management Group, 2004.

[Group 06] Object Management Group. Object constraint language - omg available specification. Relatório técnico, Object Management Group, 2006.

[Génova 02] G. Génova, J. Llorens, e V. Quintana. Digging into Use Case Relationships. Em J.-M. Jézéquel, H. Hussmann, e S. Cook, editores, *UML 2002*, volume 2460 de *Lecture Notes in Computer Science*, páginas 115–127. Springer-Verlag, 2002.

[Harel 87] D. Harel. *Statecharts: A Visual Formalism for Complex Systems*, páginas 231–274. Science of Computer Programming. North-Holland, 1987.

[Harel 90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Tauring, e M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4), 1990.

[Helke 97] S. Helke, T. Neustupny, e T. Santen. Automating test case generation from z specifications with isabelle. Em J.P. Bowen, M.G. Hinchey, e D. Till, editores, *ZUM'97: The Z Formal Specification Notation*, número 1212 em LNCS, páginas 52–71. Springer-Verlag, 1997.

[Hoare 74] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[Hoare 85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hutt 94] A. Hutt. *Object Analysis and Design: Description of Methods*. John Wiley & Sons, 1994.

[Isoda 03] S. Isoda. A Critique of UML's Definition of the Use Case Class. Em P. Stevens, J. Whittle, e G. Booch, editores, *UML 2003*, volume 2863 de *Lecture Notes in Computer Science*, páginas 280–294. Springer-Verlag, 2003.

[Jacobson 92] Ivar Jacobson. *Object-Oriented Software Engineering - a Use Case Driven Approach*. Addison-Wesley, 1992.

[Jacobson 99] Ivar Jacobson, Grady Booch, e James Rumbaugh. *The Unified Software Development Process*. Addinson-Wesley, 1999.

[Jacobson 05] Ivar Jacobson e Pan.Wei Ng. *Aspect-Oriented Software Development with Use Cases*. Addison Wesley, 2005.

[Jayaratna 94] Nimal Jayaratna. *Understanding and Evaluating Methodologies - NIMSAD: A Systemic Framework*. McGraw Hill, 1994.

[Johannisson 05] Kristofer Johannisson. *Formal and Informal Software Specifications*. Tese de Doutoramento, Chalmers University of Technology, 2005.

[Jones 86] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1986.

[Jones 88] Geraint Jones e Michael Goldsmith. *Programming in Occam2*. Prentice-Hall, 1988.

[Kiczales 97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, e John Irwin. Aspect-oriented programming. Em Mehmet Akşit e Satoshi Matsuoka, editores, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, páginas 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[Kleppe 01] Anneke Kleppe e Jos Warmer. Unification of static and dynamic semantics of uml. Relatório técnico, 2001.

[Kleppe 03] Anneke G. Kleppe, Jos Warmer, e Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[Krasner 88] G. Krasner e S. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[Language 97a] Unified Modeling Language. Uml metamodel. Relatório técnico, Rational Software, 1997.

[Language 97b] Unified Modeling Language. Uml semantics. Relatório técnico, Rational Software, 1997.

[Larman 05] Craig Larman. *Applying UML and Patterns*. Prentice Hall, 2005.

[Legeard 02] B. Legeard, F. Peureux, e M. Utting. Automated boundary testing from z and b. Em *Proceedings of the Conference on Formal Methods Europe*, 2002.

[Martins 95] F. Mário Martins. *Métodos Formais na Concepção e Desenvolvimento de Sistemas Interactivos*. Tese de Doutoramento, Escola de Engenharia, Universidade do Minho, 1995.

[Martins 98] F. M. Martins. Camila: uma Abordagem Moderna e Rigorosa para a Engenharia Informática. *Anais de Engenharia e Tecnologia Electrotécnica*, II(5), 1998.

[Maskeri 02] Girish Maskeri, James Willans, Tony Clark, Andy Evans, Stuart Kent, e Paul Sammut. A pattern based approach to defining translations between languages. Em *4 th Workshop on Rigorous Object Oriented Methods*. King's College, March 2002.

[Mellor 02] Stephen J. Mellor e Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison Wesley, 2002.

[Mencl 04] Vladimír Mencl. *Use Cases: Behavior Assembly, Behavior Composition and Reasoning*. Tese de Doutoramento, Charles University in Prague, Faculty of Mathematics and Physics, 2004.

[Meyer 92] Bertrand Meyer. Applying design by contract. *Computer*, 25(10):40–51, Oct 1992.

[Mills 80] H. D. Mills, D. O'Neill, R.C. Linger, M. Dyer, e R. Quinnan. The management of software engineering. *IBM Sys. Journal*, 24(2), 1980.

[Molkken 03] Kjetil Molkken e Magne Jrgensen. A review of surveys on software effort estimation. Em *ISESE '03: Proceedings of the 2003 International Symposium on Empirical Software Engineering*, página 223, Washington, DC, USA, 2003. IEEE Computer Society.

[Moreira 99] Ana Moreira e João Araújo. Generating object-z specifications from use cases. Em Joaquim Filipe, editor, *Enterprise Information Systems*, páginas 43–50. Kluwer Academic Publishers, 1999.

[Morris 96] D. Morris, G. Evans, P. Green, e C. Theaker. Object Oriented Computer Systems Engineering. Em *Applied Computing*. Springer-Verlag, 1996.

[Mylopoulos 99] John Mylopoulos, Lawrence Chung, e Eric Yu. From object-oriented to goal-oriented requirements analysis. *Commun. ACM*, 42(1):31–37, 1999.

[Nunes 00] Nuno Jardim Nunes e João Falcão e Cunha. Towards a uml profile for interaction design: the wisdom approach. Em *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, páginas 101–116, 2000.

[Nunes 01] Nuno Jardim Nunes e João Falcão e Cunha. Wisdom — A UML based architecture for interactive systems. *Lecture Notes in Computer Science*, 1946, 2001.

[Odell 00] James Odell, H. Van Dyke Parunak, e Bernhard Bauer. Extending uml for agents. Em *AOIS Workshop - AAAI*, 2000.

[Oliveira 95] J.N. Oliveira. Fuzzy object comparison and its application to a self-adaptable query mechanism. Em *IFSA'95*, volume I, páginas 245–248, 22–28 July 1995. Proc. of the 6th International Fuzzy Systems Association World Congress, S. Paulo, Brazil.

[Övergaard 98] Gunnar Övergaard e Karin Palmkvist. A formal approach to use cases and their relationships. Em Jean Bézivin e Pierre-Alain Muller, editores, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, páginas 309–317. Springer-Verlag, 1998.

[Paige 02] R. Paige, J. Ostroff, e P. Brooke. Checking the consistency of collaboration and class diagrams using pvs, 2002.

[Paternò 99] F. Paternò. *Model Based Design and Evaluation of Interactive Applications*. Applied Computing. Springer Verlag, 1999.

[Paternò 00] F. Paternò. ConcurTaskTrees and UML: how to marry them? Position paper at TUPIS'00 – a UML 2000 Workshop. York, UK, October 2000.

[Paulk 94] Mark Paulk. A comparison of iso 9001 and the capability maturity model for software. Relatório técnico, Software Engineering Institute, Carnegie Mellon, 1994.

[Pressman 97] R. S. Pressman. *Software Engineering: A Practioner's Approach*. McGraw-Hill, 4ª edição, 1997.

[Raistrick 04] Chris Raistrick, Paul Francis, John Wright, Colin Carter, e Ian Wilkie. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.

[Ribeiro 98] António Nestor Ribeiro. Uma arquitectura software para a prototipagem de sistemas interactivos. Tese de Mestrado, Dep. Informática, Universidade do Minho, 1998.

[Ribeiro 07] António Nestor Ribeiro, José C. Campos, e F. Mário Martins. Integrating hci in a software engineering course. Em *Proceedings of the HCI Educators 2007 Conference*, número ISBN: 978-972-789-227-3, páginas 49–58, 2007.

[Robertson 06] Suzanne Robertson e James C. Robertson. *Mastering the Requirements Process*. Addison Wesley, 2006.

[Rosenberg 07] Doug Rosenberg e Matt Stephens. *Use Case Driven Object Modeling with UML*. APress, 2007.

[Roussev 03] Boris Roussev. Generating ocl specifications and class diagrams from use cases: A newtonian approach. Em *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9*, página 321.2, Washington, DC, USA, 2003. IEEE Computer Society.

[Rumbaugh 91] James Rumbaugh, M. Blaha, W. Premerlani, e F. Eddy. *Object-Oriented Modelling and Design*. Prentice-Hall, 1991.

[Rumbaugh 98] James Rumbaugh, Ivar Jacobson, e Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[Rumbaugh 05] James Rumbaugh, Ivar Jacobson, e Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2ª edição, 2005.

[Ryser 99] J. Ryser e M. Glinz. A scenario-based approach to validating and testing software systems using statecharts. Em *12th International Conference Software and Systems Engineering and their Applications*, Dec 1999.

[Ryser 00] J. Ryser e M. Glinz. Using dependency charts to improve scenario-based testing. Em *17th International Conference Testing Computer Software*, Jun 2000.

[Schauerhuber 07] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer, e G. Kappel. A survey on aspect-oriented modeling approaches. Relatório técnico, Vienna University of Technology, 2007.

[Selic 04] Bran Selic. On the Semantic Foundations of Standard UML 2.0. Em M. Bernardo e F. Corradini, editores, *Formal Methods for the Design of Real-Time Systems*, volume 3185 de *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[Selic 06] Bran Selic. Speed Development with UML 2.0. http://www.devx.com/ibmrational, 2006.

[Sendall 00] Shane Sendall e Alfred Strohmeier. From Use Cases to System Operation Specifications. Em *UML'2000 - The Unified Modeling Language: Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000*, volume 1939, páginas 1–15, York, UK, October 2000. Springer-Verlag.

[Sendall 02] Shane Sendall. *Specifying Reactive System Behavior*. Tese de Doutoramento, École Polytechnique Fédérale de Lausanne, 2002.

[Sinnig 07] D. Sinnig, P. Chalin, e .F. Khendek. Consistency between task models and use cases. Em *Proceedings of Design, Specification and Verification of Interactive Systems 2007, Salamanca, Spain, 2007*, 2007.

[Software 98] R. Software. The rational unified process, 1998.

[Spivey 89] J. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1989.

[Straeten 04] Ragnhild Van Der Straeten, Viviane Jonckers, e Tom Mens. Supporting model refactorings through behaviour inheritance consistencies. Em Thomas Baar, Alfred Strohmeier, Ana Moreira, e Stephen J.

Mellor, editores, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, volume 3273 de *LNCS*, páginas 304–319. Springer, 2004.

[Strohmeier 04] Alfred Strohmeier, Thomas Baar, e Shane Sendall. Applying Fondue to Specify a Drink Vending Machine. *Electronic Notes in Theoretical Computer Science, Proceedings of OCL 2.0 Workshop at UML'03*, 102:155–175, 2004.

[Tahat 01] L. Tahat, B. Vaysburg, B. Koreland, e A. Bader. Requirement-based automated black-box test generation. Em *Proceedings of the 25th Annual International Computer Software and Applications Conference*, 2001.

[Union 02] International Telecommunications Union. ITU Recommendation Z.100: Specification and Description Language. Relatório técnico, ITU, 2002.

[vEB98] P. van Emde Boas. Formalizing UML: Mission Impossible. Em *Thirteenth Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98)*, Vancouver, Canada, 1998.

[Warmer 04] Jos Warmer e Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2004.

[Wegmann 00] Alain Wegmann e Guy Genilloud. The roles of roles in use case diagrams. Em Andy Evans, Stuart Kent, e Bran Selic, editores, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939, páginas 210–224. Springer, 2000.

[Yourdon 91] E. Yourdon. *Modern Structured Analysis*. Prentice-Hall, 1991.

Glossário

Abordagem operacional Forma de modelação em que se utilizam especificações executáveis.

Actividade Elemento de trabalho que se deve efectuar em determinado tempo. Em UML, significa uma operação mais demorada, cuja execução pode ser interrompida.

Actor Entidade externa a um sistema mas que com ele interage.

Agregação Forma especial de associação que especifica uma relação entre o agregado e os componentes que, física ou logicamente, contém.

Análise Fase do processo de desenvolvimento de um sistema em que se produz um modelo abstracto para descrever os aspectos fundamentais do domínio de aplicação.

Associação Relação estrutural que descreve as ligações entre classes de objectos.

Caso de uso Descrição de sequências de acções que um sistema desempenha e que produz um resultado relevante para alguns actores desse sistema.

Cenário Sequência específica de acções que ilustram o comportamento dum sistema aquando da invocação de um caso de uso.

Composição Forma mais restrita de agregação, em que os componentes dum agregado são criados e destruídos por este, mas que não podem ser partilhados por outras entidades.

Concepção Fase do processo de desenvolvimento em que, com base no modelo obtido na fase de análise, é criado um modelo arquitectural que especifica os componentes que realizam uma determinada solução para o sistema.

Desenvolvimento Conjunto das fases do ciclo de vida responsáveis pela construção do sistema, na vertente de produção de artefactos em código de uma determinada linguagem de programação.

Equipa de projecto Conjunto de engenheiros de software e gestores de projecto que efectuam as fases do processo de software, desde a análise até à instalação e posterior manutenção.

Estado Período de tempo durante o qual um sistema exibe um tipo específico de comportamento; conjunto de variáveis que retratam o modo em que um sistema se encontra.

Implementação Fase do processo de em que se faz a transição dos ambientes de desenvolvimento para o ambiente de instalação.

Mensagem Forma de comunicação entre objectos que pressupõe a execução duma dada operação.

Meta-modelo (modelo dum modelo) Conjunto de elementos de composição, funcionais ou estruturais, e de regras de composição que permitem construir um modelo para um dado sistema.

Metodologia Abordagem metódica para o desenvolvimento dum sistema, através da selecção dum modelo do processo e dum conjunto de métodos (ou técnicas) a serem aplicados.

Modelo Representação conceptual dum sistema, à luz dum determinado meta-modelo.

Modelo da Aplicação Conjunto de entidades que pertencem ao domínio do sistema que vai ser construído. Podem ser concretizações de entidades do modelo de domínio ou então apenas artefactos da tecnologia de programação utilizada.

Modelo de Domínio Conjunto de entidades do mundo do negócio organizadas entre si de uma forma que torna explíctas as regras de negócio existentes.

Modelo de processo Esquema que organiza (ordena) e relaciona a forma como as várias fases e tarefas devem ser prosseguidas ao longo de vida do sistema. A função principal dum modelo do processo é determinar a ordem das fases envolvidas durante a existência dos sistemas e estabelecer os critérios de transição para progredir entre fases.

Método Conjunto de actividades que organizam a execução duma dada fase do ciclo de vida; modo de prosseguir com uma dada fase de desenvolvimento. Implementação duma dada operação duma classe.

Objecto Entidade individual, real ou abstracta, com uma função bem definida no domínio do problema e que pode ser reconhecida pelos dados que incorpora, pelo comportamento que exibe e pelo processamento que desempenha.

Padrão Solução comum para um problema típico num dado contexto.

Projectista Indivíduo responsável pelo projecto (ou por uma parte dele) num dado sistema.

Requisito Intenção que o cliente expressa, de forma mais ou menos formal, de uma determinada funcionalidade que pretende ver contemplada no sistema.

Sistema Colecção de componentes inter-relacionados que actuam como um todo, para atingir um determinado objectivo. É o observador do sistema que define a fronteira deste como o seu ambiente, o que torna a definição de sistema não intrínseca a este, mas dependente do seu observador, ou seja, dos objectivos particulares em cada situação.

Teste Fase que pode decorrer em paralelo com o desenvolvimento dum sistema, a fim de se tentar encontrar todas as falhas daquele.

UML (Unified Modeling Language) Linguagem para expressar a funcionalidade, a estrutura e as relações de sistemas complexos.