Pedro Miguel Gonzalez de Abreu Ribeiro

O desenvolvimento de software na perspectiva das representações e das transformações entre representações

Universidade do Minho

1994

Pedro Miguel Gonzalez de Abreu Ribeiro

O desenvolvimento de software na perspectiva das representações e das transformações entre representações

Tese submetida à Universidade do Minho para obtenção do grau de Mestre em Informática, especialização em Informática de Gestão, elaborada sob a Orientação do Doutor João Álvaro Carvalho.

Universidade do Minho

1994

Resumo

Actualmente o processo de desenvolvimento de software é visto numa perspectiva de engenharia. Neste contexto, a evolução do processo segue um conjunto de fases, que permitem definir o que vai ser desenvolvido, definir como desenvolver e finalmente desenvolver e acompanhar o software. Verifica-se ainda que um modelo do processo de desenvolvimento de software define uma estrutura de etapas, que incluem as fases atrás referidas, permitindo cobrir aspectos técnicos e/ou de gestão do desenvolvimento de software. Considera-se que um modelo do processo, permite dar ênfase a determinados aspectos do desenvolvimento de software que se pretende analisar.

No âmbito deste trabalho é utilizado um modelo de representações / transformações, sendo o processo de desenvolvimento de software visto como um conjunto de representações suportadas por técnicas de representação e transformações que permitem a passagem entre duas representações.

Nesta dissertação é feita uma revisão de técnicas de representação e transformações entre representações. Procura-se identificar os tipos de representações e de transformações mais convenientes, reduzindo assim a incerteza dos intervenientes no processo e justificando a utilização de determinadas abordagens ao desenvolvimento de software.

Abstract

In our days, the process of software development is seen through an engineering perspective. In this context, the evolution of the process follows a certain set of phases, which allow the defining of what is to be developed, the defining of how to develop it, and finally the developing and maintenance of the software. One can notice that a model of the process of software development defines a structure organized in different steps, which include the phases mentioned above, allowing the study of technical aspects and/or of software development management. One considers that a model of the process emphasizes certain aspects of software development which one aim to analyse.

In pursuing this goal, a model of representations / transformations is used, and thus, the software development process is seen as a set of representations based on representation techniques and transformations which allow passing from one representation to another.

With this dissertation a revision of representation techniques and transformations between representations is made. One attempts at an identification of the most convenient representations and transformations, thus reducing the uncertainty of those people involved in the process and justifying the use of certain software development approaches.

Agradecimentos

Ao meu orientador, Professor João Álvaro Carvalho, pelos comentários, críticas e diversas sugestões dadas, pela disponibilidade sempre demonstrada, pelo seu empenho e dedicação neste trabalho.

A todos os familiares, muito especialmente à minha esposa Nany, pelo apoio e motivação com que sempre me acompanharam.

À Doutora Helena Moreira, pela colaboração prestada na revisão de versões preliminares deste texto

Finalmente, agradeço a todos os que de alguma forma contribuiram para a realização deste trabalho.

Pedro Miguel Gonzalez de Abreu Ribeiro

à Nany

Conteúdo

Lista de Figuras

(esta página foi deixada propositadamente em branco)

1. INTRODUÇÃO

O presente trabalho analisa o processo de desenvolvimento de software na perspectiva das representações utilizadas na evolução do processo e das transformações entre essas representações. Pretende-se observar o desenvolvimento de software como um conjunto de representações suportadas por técnicas de representação e transformações que permitem a 'passagem' entre duas representações, numa tentativa de verificar que combinações entre representações e transformações serão mais convenientes para o desenvolvimento de software.

O desenvolvimento de software, sendo uma área de grandes investimentos, apresenta em simultâneo uma fraca satisfação dos clientes. A tendência actual é considerar o processo de desenvolvimento de software numa perspectiva de engenharia, isto é, analisar o software como um produto. Nesta perspectiva, qualquer projecto de engenharia tem necessariamente que seguir um conjunto de passos, que permitem definir o que vai ser desenvolvido, definir como desenvolver e finalmente desenvolver e acompanhar o produto. Verifica-se que nas áreas tradicionais da engenharia, estes passos são suportados por um conjunto de ferramentas, com uma forte base científica e devidamente consolidadas pela sua utilização prática num período de tempo relativamente grande.

Neste contexto, mas em relação ao desenvolvimento de software, utilizam-se também quatro fases para caracterizar o processo, embora com designações específicas, nomeadamente: análise dos requisitos do software (*definir o quê*), concepção do software (*definir como*), implementação do software (*construir o produto*) e operação e manutenção (*acompanhar o produto*). Os produtos resultantes de cada uma das três primeiras fases,

denominam-se respectivamente: especificação dos requisitos do software, especificação do software e software

Um modelo do processo de desenvolvimento de software define uma estrutura de etapas que inclui com mais ou menos detalhe as fases atrás referidas, permitindo cobrir aspectos técnicos e de gestão do desenvolvimento de software. Os modelos do processo de desenvolvimento de software suportam diversas perspectivas, dependendo do tipo de informação necessária e do nível de detalhe do modelo. Permitem dar ênfase aos aspectos a que queremos dar relevância. Pode-se, por exemplo, visualizar o desenvolvimento de software como: i) uma actividade a ser planeada, controlada e organizada; ii) um processo colaborativo de resolução de problemas; iii) um processo de construção, transformação e modificação de representações.

No âmbito deste trabalho, interessa analisar o processo de desenvolvimento de software, na perspectiva das diversas representações possíveis e das transformações realizadas na evolução do processo. A figura 1, apresenta de um modo esquemático e simplificado as noções principais, intervenientes nesta perspectiva.

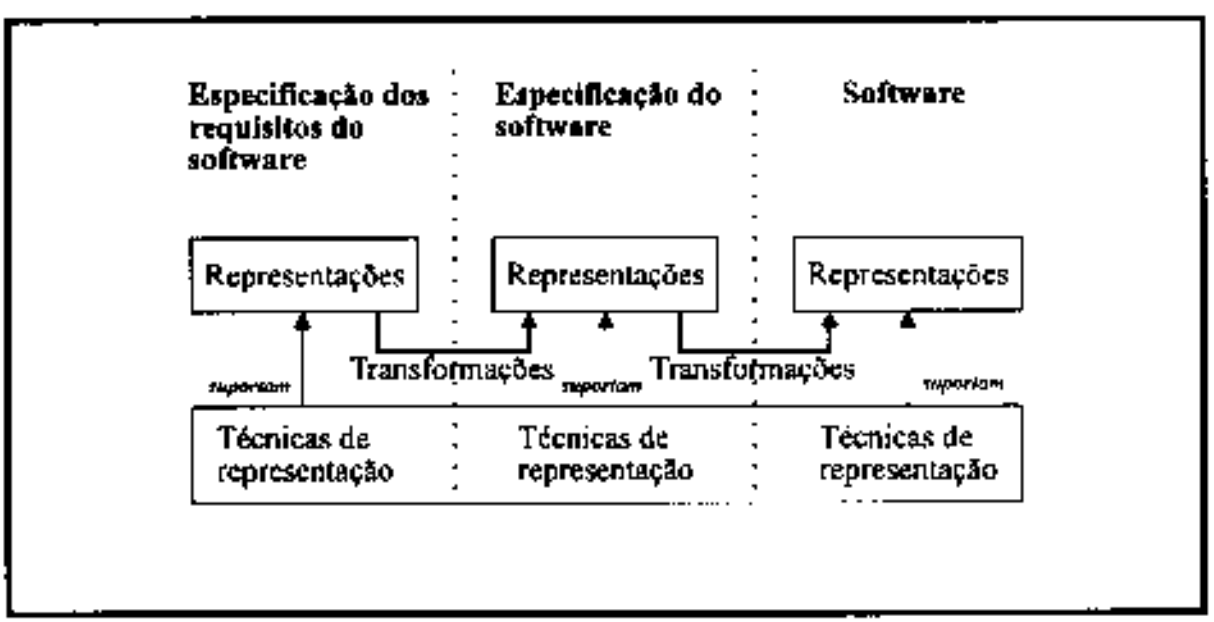


Figura 1 – Conceitos principais do modelo de representações / transformações

Considera-se que as **técnicas de representação** definem uma notação utilizável no desenvolvimento das representações, isto é, apresentam uma sintaxe e semântica precisa para cada um dos componentes da notação. Uma **representação** corresponde a uma determinada

organização dos componentes definidos pela técnica de representação. Uma **transformação** é uma passagem entre duas representações.

Pretende-se através deste modelo do processo de desenvolvimento de software verificar quais as representações e transformações mais convenientes, reduzindo assim a incerteza dos intervenientes no processo e justificando também a utilização de determinadas abordagens ao desenvolvimento de software e tipos de linguagens de programação. Este objectivo principal implica naturalmente a revisão de técnicas de representação e transformações.

Dividiu-se esta dissertação em duas partes principais. A primeira parte corresponde ao capítulo 2 e apresenta e define os conceitos essenciais sobre o processo de desenvolvimento de software, justificando o modelo de representações / transformações seleccionado. Na segunda parte, constituida pelos capítulos 3 e 4 apresenta-se uma revisão e classificação de representações e transformações.

Em seguida, descreve-se sucintamente o conteúdo de cada capítulo.

O ***segundo capítulo*** tem como objectivo apresentar em pormenor o processo de desenvolvimento de software. Assim, define-se inicialmente o estado actual da engenharia de software comparando-o com as fases de qualquer projecto de engenharia.

Define-se em seguida a terminologia e conceitos principais do processo de desenvolvimento de software, utilizados no trabalho subsequente. O capítulo prossegue, apresentando os modelos do processo de desenvolvimento de software mais usuais e definindo o papel das ferramentas CASE na evolução do software.

Finalmente, apresentam-se as diferentes perspectivas inerentes ao processo de desenvolvimento de software, definindo-se a perspectiva de gestão do processo e o modelo de representações / transformações. Este último modelo, com algumas alterações, foi seleccionado como modelo adoptado do processo de desenvolvimento de software, servindo como base dos capítulos seguintes.

O ***terceiro capítulo***, apresenta uma revisão de técnicas de representação. Inicialmente define-se o enquadramento dos métodos de desenvolvimento de software no modelo de representações / transformações. Em seguida, descrevem-se algumas técnicas de especificação

dos requisitos do software e classificam-se quanto à sua forma e conteúdo. As técnicas de especificação do software são também apresentadas e classificadas em relação à sua forma e conteúdo. Finalmente as linguagens de programação são descritas globalmente e classificadas em relação ao paradigma de programação suportado e às áreas de aplicação adaptadas.

O *quarto capítulo*, tem como objectivo descrever e classificar as transformações entre representações. Nesta perspectiva dividiram-se as transformações em três grupos: completas, parciais e equivalentes, dependendo da evolução no processo de desenvolvimento de software e das representações inicial e final. As transformações completas são descritas em pormenor, devido ao seu carácter evolutivo e ao facto de se caracterizarem por representações diferentes no inicio e fim do processo.

O capitulo termina com a apresentação das três estratégias principais de desenvolvimento de software: abordagem convencional não formal, abordagem convencional formal e abordagem orientada aos objectos. Utiliza-se em seguida os conceitos de transformação homogénea e não homogénea, para caracterizar globalmente a 'passagem' entre as representações características das três estratégias propostas.

No *quinto capítulo*, desenvolvem-se algumas conclusões, baseadas no trabalho apresentado.

2. PROCESSO DE DESENVOLVIMENTO DE SOFTWARE

2.1. Engenharia de Software

A definição de Engenharia de Software e o próprio termo, suscitam alguma controvérsia na comunidade informática. Considera-se contudo, que o desenvolvimento de software é uma disciplina de engenharia pela sua natureza inerente, embora o não seja pela sua prática corrente (McDermid 1991), (Brooks 1987).

Para Pfleeger (1991, p 6) "a engenharia de software é a aplicação dos computadores como ferramentas para resolver problemas". Esta definição obtém-se pela análise de outras disciplinas de engenharia, isto é, considera-se as ciências de computação responsáveis pela investigação e definição de teorias e ferramentas que serão aplicadas pela engenharia de software para resolver problemas.

Shaw (1990, p.17) apresenta outra perspectiva, defendendo que, "a engenharia de software ainda não é uma verdadeira disciplina de engenharia, embora possua esse potencial. O problema fundamental deve-se á não existência de uma base científica coerente".

Se considerarmos as características específicas do Software, analisado como um produto e a definição de Engenharia, poder-se-á definir Engenharia de Software, como sendo

"A ciência e arte de especificar, conceber, implementar e manter - economicamente e dentro do prazo - programas, documentação e procedimentos operacionais, através dos quais os computadores se tornam úteis aos homens" (McDermid 1991,p. II/1).

Neste contexto "arte" deve ser entendida como o processo criativo e a expressão "economicamente e dentro do prazo" como englobante dos aspectos ligados à gestão.

De um modo mais sintético, pode-se afirmar que a engenharia de software é uma estratégia para produzir software de qualidade (Basili e Musa 1991). Entendendo qualidade de um ponto de vista dos utilizadores, dos encarregados da manutenção e dos recursos utilizados, o software deve (Pfleeger 1991).

- cumprir os requisitos exigidos;
- utilizar os recursos de hardware de um modo correcto e eficiente;
- ser de fácil aprendizagem e utilização;
- ser de fácil manutenção.

2.2. Fases de um projecto de engenharia

Independentemente do método, conceitos, técnicas e ferramentas utilizadas pela equipa de desenvolvimento, qualquer projecto de engenharia tem necessariamente que passar por uma série de fases (Shemer 1987):

1. Definir o que deve ser desenvolvido;
2. Definir como desenvolver;
3. Desenvolver o produto;
4. Acompanhar o produto desenvolvido.

A primeira fase refere-se à determinação e representação do sistema pretendido pelo cliente e necessita de ferramentas de representação simples, de modo a tornar o diálogo com o cliente possível. Note-se que o objectivo principal é determinar a natureza do problema do cliente, pelo que, a discussão de qualquer solução é prematura, até o problema estar claramente definido (Pfleeger 1991).

O resultado desta primeira fase deve ser uma descrição completa do sistema e do ambiente, isto é, dos requisitos do cliente. Esta descrição deve incluir aspectos como a funcionalidade, restrições funcionais, alguns aspectos não funcionais (por exemplo, restrições à concepção ou requisitos de desempenho) e o próprio planeamento e gestão do projecto (Pressman 1992).

Na segunda fase o sistema é visto na perspectiva da equipa de desenvolvimento e não do cliente. É descrito o modo como a funcionalidade desejada será obtida e todas as estruturas necessárias ao software.

Utilizando a definição de Taylor (1959), referindo-se à concepção de um produto genérico, "é o processo de aplicar várias técnicas e princípios com o objectivo de definir um dispositivo, um processo ou um sistema com detalhe suficiente que permita a sua realização física". Em termos mais específicos, consiste em conceber a arquitectura, o interface, os dados e os procedimentos necessários para satisfazer os requisitos expressos na fase anterior

A terceira fase, consiste na implementação, isto é, a construção do produto e a sua verificação e validação. É a tradução dos documentos obtidos na etapa anterior para uma linguagem de programação. Embora, possa parecer uma etapa com pouca influência na qualidade final do software é necessário estar consciente que as características específicas da linguagem de programação bem como o desempenho "criativo" dos próprios programadores pode influenciar decisivamente a qualidade final do Software.

A quarta fase é a actividade de manutenção, necessária em qualquer produto, definindo-se como o conjunto de actividades necessárias para manter e melhorar o produto, depois de este ser aceite e instalado no cliente. No caso do software, normalmente considera-se dividida em quatro tipos (Pressman 1992): correctiva, perfectiva, adaptativa e preventiva.

Verifica-se que qualquer projecto de desenvolvimento de software deve passar pelas quatro fases apresentadas anteriormente, mas considera-se fundamental a distinção entre estas fases (inerentes a todos os projectos) e o modelo do processo (paradigma) de desenvolvimento de software.

2.3. Terminologia e conceitos principais do processo de desenvolvimento de software

Apresenta-se em seguida a terminologia e definições adoptadas para descrever o processo de desenvolvimento de software.

2.3.1. Fase de análise dos requisitos do software

A etapa inicial do processo de desenvolvimento de software tem um papel crucial para o sucesso do software resultante (Fraser, Kumar e Vaishnavi 1991), (Dorfman 1990), (Yadav et al. 1988), (Shemer 1987). Brooks (1987, p.17) afirma mesmo que "a parte mais complexa no processo de desenvolvimento de software é decidir com precisão o que vamos construir. Nenhuma outra parte do trabalho conceptual é tão difícil como o estabelecimento dos requisitos técnicos detalhados, incluindo todos os interfaces com pessoas, máquinas e outros sistemas de software. Nenhuma outra parte do desenvolvimento invalida tanto o sistema resultante se realizada de modo incorrecto. Nenhuma outra parte é tão difícil de rectificar mais tarde".

Existem diversas normas internacionais sobre a primeira etapa do processo de desenvolvimento de software, por exemplo, DoD[1]-STD-2167A (1988), ESA[2] PSS-05-0 (1987) ou ANSI/IEEE Std. 830 (1984). Contudo as opiniões divergem sobre a terminologia a adoptar e sobre as actividades e resultados desta etapa. Para uniformizar os conceitos utilizados, adoptamos os termos **análise dos requisitos do software**[3] **e especificação dos requisitos do software**[4] (Pressman 1992), (McDermid 1991), (DoD-STD-2167A 1988), (ANSI/IEEE Std. 830-1984) para referenciar respectivamente a primeira etapa do processo de desenvolvimento de software e os documentos resultantes desta etapa.

A análise dos requisitos do software define-se como: "1. O processo de estudo das necessidades do utilizador para obter a definição dos requisitos do software. 2. A verificação dos requisitos do software" (ANSI/IEEE Std. 729-1983). Como resultado da análise dos requisitos do software obtém-se a especificação dos requisitos do software, isto é, "um documento que descreve claramente e com precisão cada um dos requisitos essenciais

[1] *U.S. Department of Defense.*
[2] *European Space Agency.*
[3] *Software Requirements Analysis.*
[4] *Software Requirements Specification.*

(funções, desempenho, restrições à concepção e atributos de qualidade) do software e dos interfaces externos. Cada requisito deve ser definido de tal modo que a sua obtenção possa ser verificada objectivamente por um método prescrito, por exemplo, inspecção, demonstração, análise ou teste" (ANSI/IEEE Std. 830-1984).

Vários autores utilizam termos diferentes para referenciar os dois conceitos definidos. Por exemplo, "análise dos requisitos do software" é apresentada como análise do problema (Davis 1990), análise (Freeman 1983), definição dos requisitos do software (Brackett 1990), (ESA PSS-05-0 1987) e definição do sistema (Birrel e Ould 1985), de igual modo "especificação dos requisitos do software" é apresentada como especificação dos requisitos do utilizador (BS6719 1986), documentação dos requisitos do software (ESA PSS-05-0 1987) especificação (Charette 1986) e especificação funcional (Birrel e Ould 1985).

Segundo Brackett (1990), o processo de análise dos requisitos do software é constituido pelas seguintes actividades principais:

- Identificação dos requisitos do software, consiste na obtenção dos requisitos a partir de informações dos utilizadores, dos clientes e dos especialistas no sistema;

- Identificação das restrições ao desenvolvimento do software, por exemplo, os custos máximos, o software existente e com o qual o novo software irá operar, requisitos de portabilidade, etc;

- Avaliação dos requisitos do software, uma vez que os requisitos normalmente são obtidos a partir de diversas fontes, deve-se avaliar os potenciais problemas, ordenar os requisitos por prioridades e avaliar a exequibilidade e riscos do desenvolvimento;

- Especificação dos requisitos do software;

- Comunicação dos requisitos do software às diversas audiências para revisão e aprovação;

- Preparação para a validação dos requisitos do software, isto é, estabelecer critérios e técnicas para assegurar que o software produzido satisfaz os requisitos.

Rombach (1990) descreve classes de intervenientes e as suas responsabilidades na etapa de análise dos requisitos do software. Apresenta os clientes, responsáveis pelo contrato para o desenvolvimento e pela aceitação do software final, os utilizadores finais que irão operar e utilizar o software, os analistas, responsáveis pela especificação dos requisitos do software, os gestores que tem como funções o planeamento, controlo e gestão do projecto de desenvolvimento de software e o pessoal relacionado com a verificação e validação.

Considera-se que uma especificação dos requisitos do software deve apresentar os requisitos da seguinte forma (ANSI/IEEE Std. 830-1984):

(a) Requisitos funcionais, devem especificar o modo como as entradas para o software serão transformadas em saidas. Descrevem as acções fundamentais que o software deve executar Para cada função devem ser especificados os requisitos relacionados com as entradas, o processamento e as saidas;

(b) Requisitos de desempenho, devem especificar os requisitos numéricos, tanto estáticos como dinâmicos necessários no software ou na interacção do homem com o software.

- requisitos numéricos estáticos: número de terminais a serem suportados, número de utilizadores simultâneos, número de ficheiros e registos e tamanho previsto dos ficheiros
- requisitos numéricos dinâmicos, incluem por exemplo, o número de tarefas, o número de transacções e o volume de dados a serem processados num determinado período de tempo.

(c) Restrições à concepção, inclui limitações de hardware, existência de outras normas, etc.

(d) Atributos de qualidade, por exemplo, a segurança do software, a facilidade de manutenção ou a portabilidade.

(e) Requisitos de interface externo, inclui, interfaces com os utilizadores, interfaces com hardware, interfaces com software e interfaces para comunicações.

Balzer e Goldman (1986) apresentam oito princípios necessários para o desenvolvimento de uma boa especificação de requisitos do software:

1. Separar a funcionalidade da implementação;
2. Utilizar uma linguagem de especificação orientada aos processos;
3. Incluir a descrição do sistema do qual o software é componente;
4. Incluir a descrição do ambiente em que o sistema opera;
5. A especificação deve ser um modelo cognitivo.
6. A especificação deve ser operacional;
7. A especificação deve ser evolutiva e tolerante a faltas;
8. A especificação deve ser localizada e pouco acoplada.

Considera-se que a especificação de requisitos do software deve definir correctamente todos os requisitos do software, mas não deve apresentar detalhes de concepção, verificação ou gestão do projecto. Deve permitir uma grande flexibilidade na concepção do software. Além deste aspecto, uma boa especificação dos requisitos do software deve possuir as seguintes características (ANSI/IEEE Std 830-1984):

- **Não ambígua.** Todos os requisitos definidos têm apenas uma interpretação;
- **Completa.** Tem definida as respostas para todas as classes de dados de entrada e para todas as situações;
- **Verificável.** Para cada requisito definido, existe um processo através do qual é possível verificar se o software final satisfaz o requisito;
- **Consistente.** Não existe conflito entre nenhum dos requisitos descritos;
- **Modificável.** Possui uma estrutura e estilo, que qualquer alteração aos requisitos é realizada com facilidade e consistência,
- **Investigável.** A origem de cada requisito definido deve ser clara e a sua referência para futuros desenvolvimentos deve ser simples;
- **Utilizável.** Deve ser um documento que acompanhe todas as fases do desenvolvimento do software.

Para Brackett (1990) e Rombach (1990) os requisitos do software devem ser divididos em requisitos_C (requisitos orientados ao Cliente) e requisitos_D (requisitos orientados ao Desenvolvimento).

Considera-se que o objectivo principal destes documentos (especificação dos requisitos_C e requisitos_D) é obter um acordo sobre o que vai ser produzido, contudo a sua forma é determinada pelos condicionalismos de comunicação com os diversos participantes na etapa de requisitos de software. O desenvolvimento dos requisitos_D, refinam e incrementam os requisitos_C, para representar adequadamente a informação necessária à concepção do software e sua subsequente validação realizada em relação aos requisitos iniciais (Brackett 1990).

2.3.2. Concepção do software

A segunda fase do processo de desenvolvimento de software denomina-se **concepção do software**[5], sendo esta a terminologia utilizada por grande parte dos autores e normas internacionais (Sommerville 1992), (Pressman 1992), (Ghezzi, Jazayeri e Mandrioli 1991), (ANSI/IEEE Std. 729 1983).

O objectivo desta etapa é conceber uma solução que satisfaça os requisitos do software. Para isso são formuladas e analisadas soluções alternativas e a melhor solução é seleccionada e refinada (FIPS Pub.101 1983). Esta etapa centra-se na concepção da arquitectura e dos componentes do software, dos interfaces e dos dados (Thayer e Dorfman 1990). Normalmente divide-se esta etapa na concepção da arquitectura do software e na concepção detalhada do software (McDermid 1991), (ANSI/IEEE Std 729 1983)

A *concepção da arquitectura do software* é o processo de definição dum conjunto de componentes de software e dos seus interfaces para estabelecer uma estrutura para o desenvolvimento do software (ANSI/IEEE Std 729 1983) Existem diversos sinónimos de "concepção da arquitectura do software", por exemplo, concepção preliminar do software (DoD-STD-2167A 1988) ou concepção do sistema (Pfleeger 1991).

A *concepção detalhada do software* é o processo de refinar e expandir a concepção da arquitectura do software para incorporar descrições mais detalhadas da lógica de processamento, das estruturas e definições de dados, com o objectivo de tornar a concepção suficientemente completa para ser implementada. O refinamento adicional para codificar o software deve ser mínimo (ANSI/IEEE Std. 729 1983). A concepção detalhada do software é

[5] *Software Design*.

também denominada, concepção dos programas (Pfleeger 1991) ou concepção física (Downs, Clare e Coe 1992).

A etapa de concepção do software é um processo iterativo que inclui as seguintes actividades (STARTS Guide 1987):

- Abstracção. Caracteriza-se pela utilização da operação de generalização, identificando o essencial. Segundo Wasserman (1983, p.43), "a abstracção permite que uma pessoa se concentre num problema a um determinado nível de generalização, sem se preocupar com os detalhes irrelevantes dos níveis mais baixos";

- Decomposição. Caracteriza-se pela redução de um 'objecto' num determinado número de componentes mais simples e menores;

- Elaboração. A operação de detalhar, acrescentando características a um componente;

- Tomada de decisão. Consiste na identificação de estratégias alternativas, na sua análise e selecção.

A documentação resultante da etapa de concepção do software denomina-se, **especificação do software**[6] e encontra-se dividida em, especificação da arquitectura do software e especificação detalhada do software (ESA PSS-05-0 1987), (ANSI/IEEE Std. 729 1983).

A especificação do software deve definir os objectivos da concepção, em termos de (SWAP-DIP-P100 1989):

- Apresentação do nível proposto de satisfação dos requisitos do software verificado no sistema final;

- Grau de confiança esperado, de que o sistema final execute de uma forma consistente a sua função prevista;

[6] *Software Specification.*

- Eficiência com que o sistema implementado utilizará os recursos informáticos;
- Facilidade de manutenção;
- Transparência tecnológica.

Considera-se que a concepção da arquitectura do software deve abranger uma série de componentes, tais como, módulos, dependências, dados, recursos, processos, funções e interfaces (Hester, Parnas e Utter 1981). A figura 2 apresenta os componentes característicos da concepção da arquitectura do software e os seus relacionamentos (McDermid 1991).

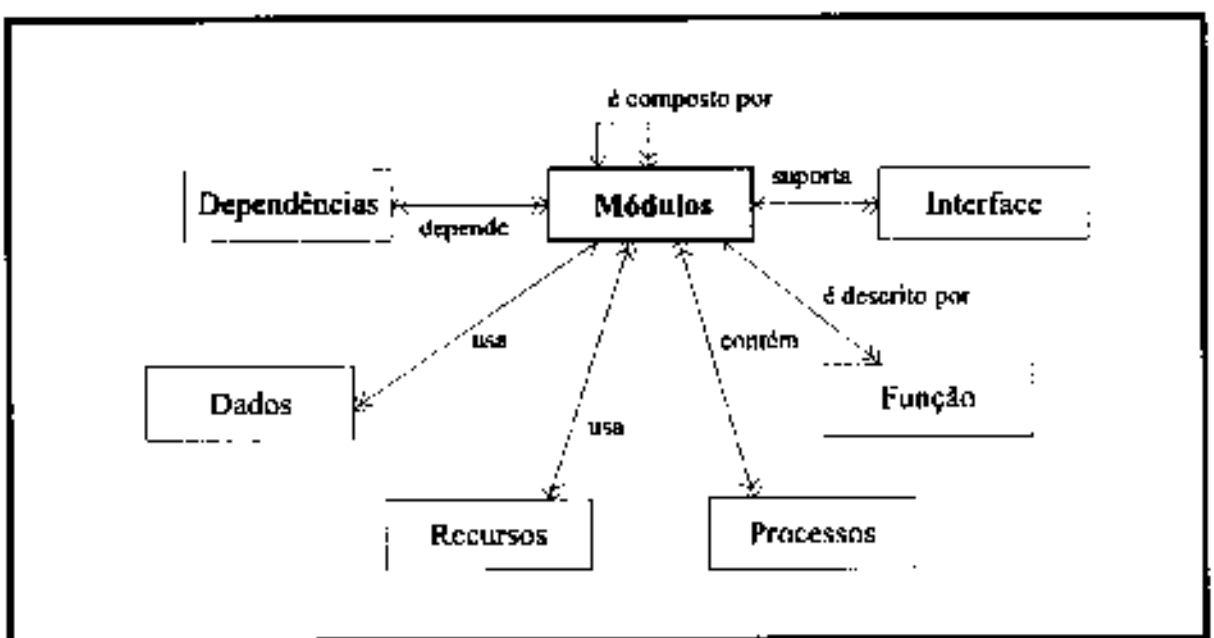


Figura 2 - Estrutura da etapa de concepção da arquitectura do software (McDermid 1991)

Assume-se que a unidade básica de decomposição / composição, utilizada na concepção da arquitectura do software é o **módulo**. Esta "entidade" terá como atributos principais, a sua identificação e uma descrição de aspectos relacionados com a qualidade, por exemplo, coerência ou reutilização (McDermid 1991). Britton e Parnas (1981) observam que, "o objectivo global da decomposição em módulos é a redução dos custos do software, permitindo que os módulos sejam concebidos e revistos independentemente". Note-se que em termos das abordagens orientadas aos objectos, os módulos servem como "recipientes" físicos em que serão declaradas as classes e objectos (Booch 1991).

McDermid (1991) descreve os vários conceitos (entidades) utilizados no diagrama de entidades e relacionamentos da figura 2:

- As "*dependências*" representam os diferentes tipos de dependências entre módulos, por exemplo, partilha da definição de dados, chamadas a procedimentos, comunicação por mensagem, etc. Exclui-se a dependência de composição[7];

- A entidade "dados", refere-se à estrutura, o relacionamento e a utilização de dados no sistema;

- Os "processos" representam a estrutura dinâmica do sistema, isto é, o seu comportamento concorrente;

- Os "recursos" referem-se aos aspectos da máquina física ou às suas características de execução, tais como, memória, tempo de *CPU*, periféricos, etc, utilizados pelos módulos e processos;

- O "interface" permite a definição dos protocolos de comunicação entre módulos, especificando o tipo e formato dos parâmetros passados. O interface com o utilizador é concebido em relação a formatos gerais e regras a seguir na implementação;

- A "função" refere-se à descrição do processamento realizado pelo módulo. Esta descrição é a principal fonte de informação utilizada na concepção detalhada do software.

A concepção detalhada de um módulo corresponde a instruções para o programador, que descrevem de um modo detalhado e técnico, as entradas, saídas e processamento do módulo. Esta especificação detalhada sobre cada módulo deve incluir, detalhes dos algoritmos utilizados, as representações e estruturas de dados e os relacionamentos entre as funções realizadas e os dados utilizados (Pfleeger 1991).

[7] A dependência composição é representada pelo relacionamento recursivo 'é composto por' na entidade módulo.

Embora não exista consenso sobre as características de "uma boa concepção", considera-se normalmente que uma boa concepção deve originar um produto de fácil manutenção. Nesta perspectiva os métodos principais para avaliar a qualidade da concepção do software são, o acoplamento, a coesão, a facilidade de compreensão e a adaptabilidade (Sommerville 1992).

Define-se acoplamento como a medida do grau de interdependência entre dois módulos. O resultado da concepção do software é tanto melhor quanto menor for o acoplamento. Pretende-se desenvolver os módulos como caixas pretas, isto é, deve ser possível trocar um módulo com um mínimo de risco de ter que trocar outro, evitando-se também a situação de aparecimento de erros em cadeia (Page-Jones 1988).

Os módulos podem ser classificados numa "escala", quanto ao acoplamento:

baixo acoplamento "bom"	• *acoplamento de dados* - comunicação por parâmetros • *acoplamento de imagem* - ligação por um grupo relacionado de dados • *acoplamento de controlo* - ligação por uma *flag* que controla a actividade do outro modelo • *acoplamento comum* - ligação através de uma área comum de dados
alto acoplamento "mau"	• *acoplamento de conteúdo* - um módulo referencia o interior de outro

Figura 3 - Tipos de acoplamento entre módulos (Myers 1975)

Segundo Sommerville (1992), os objectos possuem um fraco acoplamento entre si, uma vez que incorporam dados e operações e exteriormente apenas é reconhecido o interface dum objecto e não o modo como os serviços estão concebidos. É simples substituir um objecto por outro, basta possuírem o mesmo interface. Refira-se contudo que a herança de atributos e operações, provoca um forte acoplamento do objecto às suas superclasses.

A coesão define-se como a medida da intensidade da associação dos elementos de um módulo. O objectivo para melhorar a qualidade da concepção do software é tornar os módulos fortemente coesos (Page-Jones 1988).

Segundo Yourdon e Constantine (1978), existem sete níveis de coesão (figura 4).

forte coesão "bom"	• *funcional* - cada elemento do módulo é necessário para a realização da sua função • *sequencial* - a saída de uma actividade é a entrada da actividade seguinte • *comunicacional* - todos os elementos do módulo operam sobre os mesmos dados • *procedimental* - as actividades dentro do módulo não se relacionam, existe contudo um fluxo de controlo entre as actividades • *temporal* - o que relaciona as diversas actividades é o tempo • *lógica* - os componentes realizam funções similares (tratamento de erros, ...)
fraca coesão "mau"	• *coincidental* - não existe nenhum relacionamento funcional entre os diversos componentes do módulo

Figura 4 - Níveis de coesão entre módulos

No que se refere à concepção orientada aos objectos, reconhece-se que um objecto é, por definição, uma entidade fortemente coesa, embora, caso a funcionalidade seja obtida através de herança, existe uma redução significativa da coesão dos objectos (Pfleeger 1991). Sommerville (1992), define "coesão do objecto", como um nível específico da medida de coesão apresentada pelos objectos.

Finalmente, em termos da qualidade da concepção do software, interessa também avaliar a facilidade de compreensão dos módulos e a facilidade de adaptação da concepção. Estes dois factores são importantes na manutenção do produto final e na reutilização da especificação (Sommerville 1992).

2.3.3. Implementação e testes do software

A fase seguinte no processo de desenvolvimento de software é a fase de implementação e testes do software, tendo como resultado final o programa ou sistema pronto para instalação (FIPS-Pub-101 1983), (DoD-STD-2167A 1988).

A implementação (codificação) é o processo de transformação da especificação do software no software final. Consiste na produção de programas que devem apresentar a funcionalidade e desempenho definidos nas duas fases anteriores e utiliza como ferramentas de suporte as linguagens de programação. Pressman (1992, p.513) afirma que, "quando considerada como uma fase no processo de engenharia de software, a codificação é vista como uma consequência natural da concepção. Contudo, as características da linguagem de programação e o estilo de codificação podem afectar profundamente a qualidade e manutenção do software".

Os testes envolvem a execução do programa utilizando dados semelhantes aos dados reais que serão processados pelo programa. Pretende-se com esta actividade detectar e corrigir os erros no programa. Pode definir-se como o processo de analisar o software ou um seu componente, para detectar as diferenças entre a funcionalidade necessária e existente, avaliando assim as características do software ou do componente (ANSI/IEEE Std. 729 1983)

2.3.4. Operação e manutenção do software

A última fase do processo de desenvolvimento de software é a fase de operação e manutenção do software e envolve a utilização e acompanhamento do software (FIPS-Pub-101 1983). A manutenção é a modificação do software, depois da sua entrega, para corrigir defeitos, melhorar o desempenho e outros atributos ou para o adaptar a mudanças no ambiente (ANSI/IEEE Std 729 1983). A manutenção divide-se normalmente em quatro tipos (Pressman 1992):

- Manutenção correctiva, que inclui o diagnóstico e correcção de erros;
- Manutenção adaptativa, é o processo de modificar o software para acompanhar as alterações do ambiente;

- Manutenção perfectiva, que corresponde a alterações realizadas no software devido a novos requisitos pedidos pelos utilizadores;
- Manutenção preventiva, é o processo de alterar o software para melhorar a sua manutenção e confiança futura.

Apresenta-se em seguida a terminologia utilizada para descrever o processo de desenvolvimento de software:

Fase no processo de desenvolvimento de software	**Produto**
⇨ Análise dos requisitos do software	Especificação dos requisitos do software
⇨ Concepção do software • Concepção da arquitectura • Concepção detalhada	Especificação do software
⇨ Implementação e testes do software	Aplicação
⇨ Operação e manutenção do software	

2.4. Modelos do processo de desenvolvimento de software

O modelo do processo de desenvolvimento de software é um conjunto de etapas que inclui com mais ou menos detalhe as fases anteriores, os métodos, que definem "como" conduzir as etapas e as ferramentas, permitindo uma perspectiva técnica e de gestão ao desenvolvimento de software (Pressman 1992), (Humphrey 1989).

Segundo Curtis, Kellner e Over (1992) um modelo do processo de desenvolvimento de software deve alcançar os seguintes objectivos:

- Facilitar a compreensão e comunicação humana, para o que se torna necessário que a equipa partilhe um formato de representação comum,
- Suportar o aperfeiçoamento do processo, o que requer uma base para definir e analisar os processos;
- Suportar a gestão do processo, o que requer um processo definido a partir do qual possa ser comparado o comportamento dos projectos actuais;

- Definir e suportar as tarefas necessárias à realização do processo;
- Definir suportes automáticos para determinadas tarefas.

Os modelos do processo de desenvolvimento de software podem ser definidos no nível U (Universal), no nível M (Mundial) ou no nível A (Atómico). Considera-se que o nível U, estabelece as políticas, isto é, define uma estrutura de alto-nível e um conjunto de princípios que conduzem o comportamento global da organização; o nível M, estabelece os procedimentos necessários para implementar as políticas definidas, ou seja, especifica os pré-requisitos e resultados de cada tarefa e define quem, faz o quê, quando; o nível A, estabelece as normas que definem com precisão o modo como as tarefas são executadas (Humphrey 1989)

Segundo Curtis, Kellner e Over (1992), existem alguns factores que fazem com que o modelo do processo adoptado pelas organizações não acompanhe normalmente o seu comportamento real:

1. Perspectivas de alto-nível que não se relacionam com as actividades dos projectos;
2. Descrições imprecisas, ambíguas, incompreensíveis e irrealistas sobre as tarefas a serem realizadas nos projectos;
3. Falhas na actualização da documentação que acompanha as alterações do processo.

Apresentam-se em seguida alguns modelos do processo de desenvolvimento de software no nível Universal

2.4.1. Modelo em cascata

O modelo em cascata ("*waterfall*") ou modelo convencional do ciclo de vida de desenvolvimento de software, apresentado por Royce (1970), mas conhecido principalmente com a estrutura proposta por Boehm (1976), é o modelo do processo de desenvolvimento mais difundido e historicamente o primeiro modelo consensual do processo de desenvolvimento de software.

Este modelo apresenta o desenvolvimento de software como um conjunto de sete etapas e define a sequência e objectivos de cada etapa.

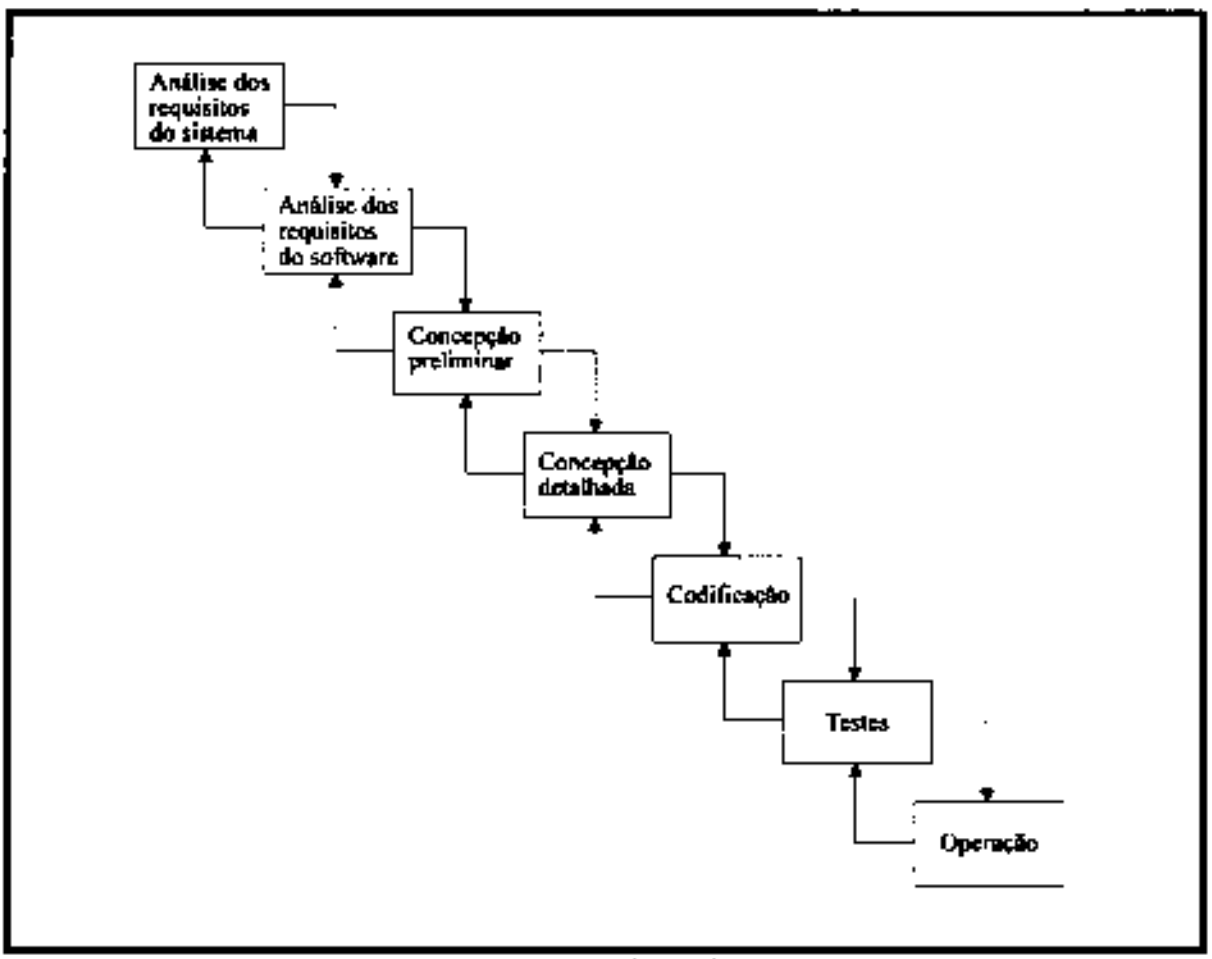


Figura 5 - Modelo em cascata do processo de desenvolvimento de software (Boehm 1976)

A maior vantagem da utilização deste modelo situa-se nas áreas de organização e controlo do projecto de software (Sage e Palmer 1990). Este modelo apresenta também outros aspectos importantes (Davis, Bersoff e Comer 1988), nomeadamente:

- aconselha a especificação completa do sistema antes da sua construção;
- encoraja a concepção do software antes da sua implementação;
- permite aos gestores do projecto um acompanhamento preciso dos seus progressos;
- aconselha a criação de um conjunto de documentos ao longo de todo o processo, que serão utilizados nas etapas finais;
- permite uma melhor estruturação e gestão de todo o processo de desenvolvimento.

Embora este modelo apresente uma perspectiva simplificada do processo de desenvolvimento de software existem diversas normas industriais e governamentais[8] baseadas no modelo em cascata.

Apesar da sua importância e validade o modelo convencional do processo de desenvolvimento de software apresenta algumas dificuldades (Humphrey 1989):

a. Não acompanha adequadamente as mudanças;
b. Assume uma sequência relativamente uniforme e ordenada de etapas de desenvolvimento;
c. Não prevê a existência de métodos que recorrem à prototipagem rápida ou linguagens avançadas.

Para resolver as deficiências apresentadas pelo modelo em cascata, foram propostos três novos modelos do processo de desenvolvimento de software (Agresti 1986): a Prototipagem, a Especificação Operacional e a Implementação Transformacional.

2.4.2. Prototipagem

A prototipagem prevê um desenvolvimento rápido de todo / parte do sistema informático para a definição e clarificação dos requisitos. O principal objectivo deste modelo é reduzir o risco e incerteza associados ao desenvolvimento de software (Pfleeger 1991).

Devem-se distinguir duas abordagens distintas associadas à Prototipagem, embora a meta comum seja sempre o desenvolvimento de um protótipo para facilitar a comunicação e o entendimento dos requisitos do utilizador (Gomaa 1990).

A *Prototipagem Rápida*, adopta o ciclo de desenvolvimento convencional, mas acrescenta-lhe uma etapa de "desenvolvimento do protótipo", que interage com a etapa de análise de requisitos até à completa clarificação dos aspectos funcionais pedidos pelo utilizador. Após a estabilização e definição dos requisitos o protótipo é abandonado e adopta-se a sequência de etapas prevista pelo modelo em cascata (concepção preliminar, concepção detalhada, ...).

[8] Por exemplo, o modelo utilizado pelo Departamento de Defesa dos Estados Unidos, DoD STD 2167A (1988), ou o ciclo de vida estruturado (Yourdon 1989).

A *Prototipagem Evolutiva* prevê o desenvolvimento de um protótipo que evoluirá até ao software final (fig 6). O objectivo é implementar inicialmente um subconjunto do sistema que será gradualmente completado. Esta abordagem é normalmente apresentada como o desenvolvimento incremental, já que o sistema final é obtido através de um processo iterativo com o utilizador (Gomaa 1990).

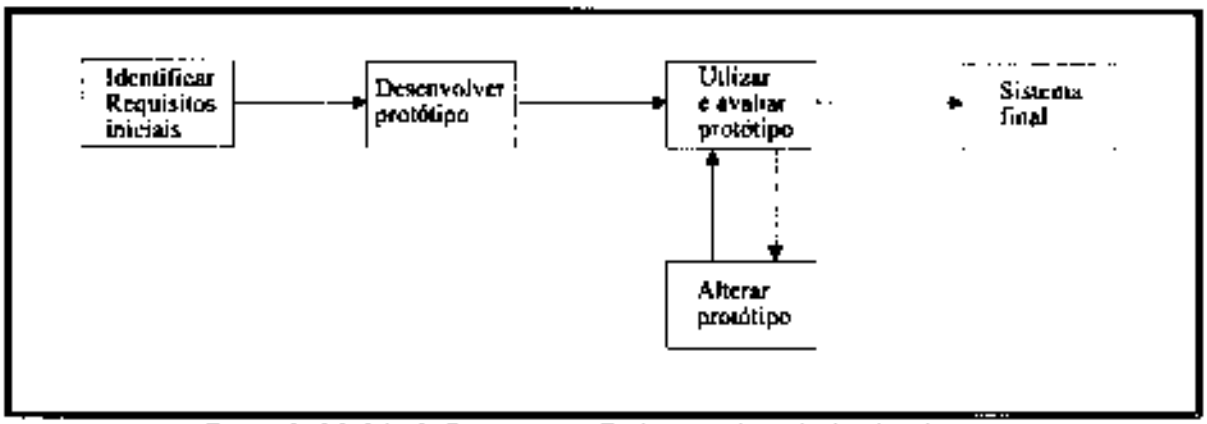


Figura 6 - Modelo da Prototipagem Evolutiva (adaptado de Alavi [1984])

As vantagens da utilização da prototipagem como modelo do processo de desenvolvimento de software, centram-se na validação e clarificação dos requisitos, verificação de determinados subsistemas mais complexos[9] e no maior envolvimento dos utilizadores na definição do sistema. A prototipagem apresenta dificuldades na alocação dos recursos, na realização de estimativas e o tempo e custo de realização da análise de requisitos aumenta em relação ao modelo convencional do processo de desenvolvimento de software (Sage e Palmer 1990).

Uma experiência curiosa realizada por Boehm, Gray e Seewaldt (1984), que compara os resultados da aplicação do modelo convencional e da prototipagem evolutiva no desenvolvimento de um produto de software de pequena complexidade, conclui que:

1. A prototipagem originou um produto com um desempenho equivalente, mas com menos linhas de código (40%) e menos esforço (45%);
2. O produto obtido através da prototipagem apresentava menor funcionalidade e robustez, mas maior facilidade de utilização e aprendizagem.
3. A utilização do modelo convencional produziu uma concepção mais coerente e um produto com maior facilidade de ser integrado.

[9] Interface com o utilizador, entradas/saídas ou mesmo utilização de determinados algoritmos específicos.

2.4.3. Especificação operacional

Outro modelo do processo de desenvolvimento de software é o modelo operacional (fig.7). Esta abordagem baseia-se na convicção da impossibilidade de diferenciar "o quê?" do "como?"[10] no desenvolvimento do software (Swartout e Balzer 1982), isto é, o desenvolvimento do software passa pela fase orientada ao problema e pela fase orientada à implementação (Agresti 1986).

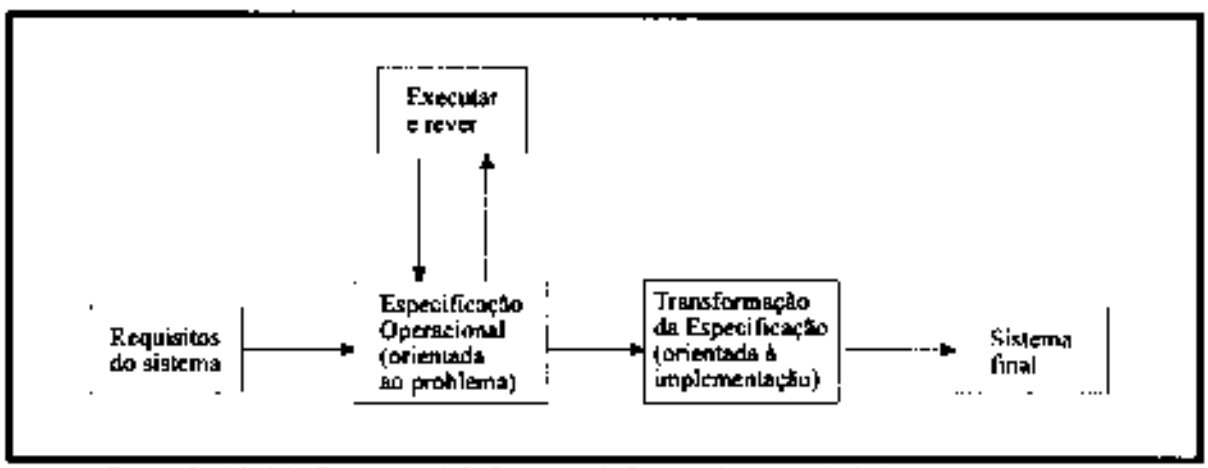


Figura 7 - Modelo Operacional do Processo de Desenvolvimento de Software (Agresti 1986)

Na primeira fase desenvolve-se uma especificação operacional, ou seja, um modelo do sistema que pode ser avaliado ou executado para gerar o comportamento do sistema. Na fase de transformação, a especificação operacional é sujeita a transformações que preservam o comportamento externo mas alteram e melhoram os mecanismos pelos quais o comportamento é produzido, obtendo-se então uma especificação orientada à implementação (Zave 1984).

Segundo Agresti (1986) o modelo operacional permite ultrapassar dois problemas existentes no modelo convencional

- Verifica-se que na etapa de concepção do software incidem demasiados aspectos não relacionados com a própria concepção, isto é, inclui decisões relacionadas com o problema, pois é necessário decompor as funções de alto nível e decisões relacionadas com a implementação, por exemplo a performance ou confiança do software.

[10] Análise de requisitos e concepção do software no modelo convencional.

- A separação artificial entre o comportamento externo (análise dos requisitos) e a estrutura interna (concepção).

Com a utilização do modelo operacional no processo de desenvolvimento de software, esperam-se vantagens devido à obtenção de um protótipo que permitirá avaliar soluções alternativas e devido à utilização de uma especificação formal, mais rigorosa. A principal desvantagem refere-se à possibilidade de serem tomadas decisões prematuras relativas à concepção do software.

2.4.4. Implementação transformacional

A implementação transformacional (fig.8) é um modelo do processo de desenvolvimento de software que se caracteriza pela utilização de um suporte automático para aplicar uma série de transformações que alteram a especificação até ao software final pretendido (Balzer 1981). Este modelo está intimamente ligado às diversas teorias sobre especificação formal de software[1].

As transformações a aplicar nas diversas fases do desenvolvimento de software incluem, a alteração das representações dos dados, selecção de algoritmos, optimização e compilação. Idealmente todas as regras de transformação preservam a correcção, de modo a que o produto final satisfaça a especificação original.

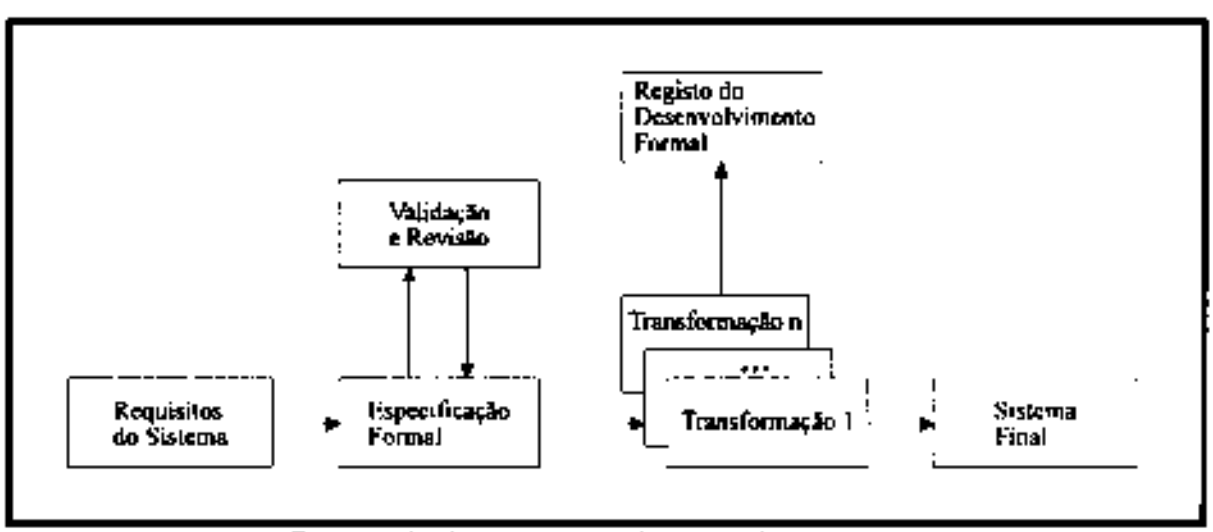


Figura 8 - Implementação transformacional (Agresti 1986)

[1] Por exemplo, Bauer (1976), Balzer, Goldman e Wile (1976), Meyer (1985) ou Jones (1990).

Segundo Agresti (1986) os benefícios esperados da utilização da implementação transformacional, situam-se em diversas áreas:

- Automatização dos processos de transformação, que reduz o esforço humano necessário no desenvolvimento de software;
- Segurança na aplicação das transformações (devido ao seu carácter formal), assegurando a correcção do processo;
- Eliminação efectiva dos testes ao produto final, sendo substituido pela verificação da especificação do programa;
- Flexibilidade de incluir o programador no "ciclo" automático de modo a seleccionar as transformações mais ajustadas.

A maior dificuldade sentida na aplicação deste modelo do processo deve-se ao facto de existirem apenas ambientes de suporte experimentais[12], isto é, considera-se uma abordagem ainda em fase de investigação. Encontra-se bastante difundido nos meios universitários mas na indústria utiliza-se apenas em pequenos programas em áreas experimentais (Ghezzi, Jazayeri e Mandrioli 1991).

Os três modelos do processo descritos - prototipagem, especificação operacional e implementação transformacional - apresentam-se como alternativos ao modelo convencional e possuem algumas semelhanças nos seus princípios básicos. Os objectivos comuns são, entregar num curto espaço de tempo um modelo executável aos utilizadores e automatizar a produção do software (Agresti 1986).

2.4.5. Modelo em espiral

Actualmente, o modelo em espiral, apresentado por Boehm (1987) é o modelo do processo de desenvolvimento de software preferido por vários autores (Sage e Palmer 1990), (Pressman 1992) e (Sommerville 1992), uma vez que capta as melhores caracteristicas do modelo convencional do processo de desenvolvimento de software, da prototipagem e acrescenta-lhe a análise do "risco".

[12] Os trabalhos de Tsai e Ridge (1988) e Bauer, et al. (1989).

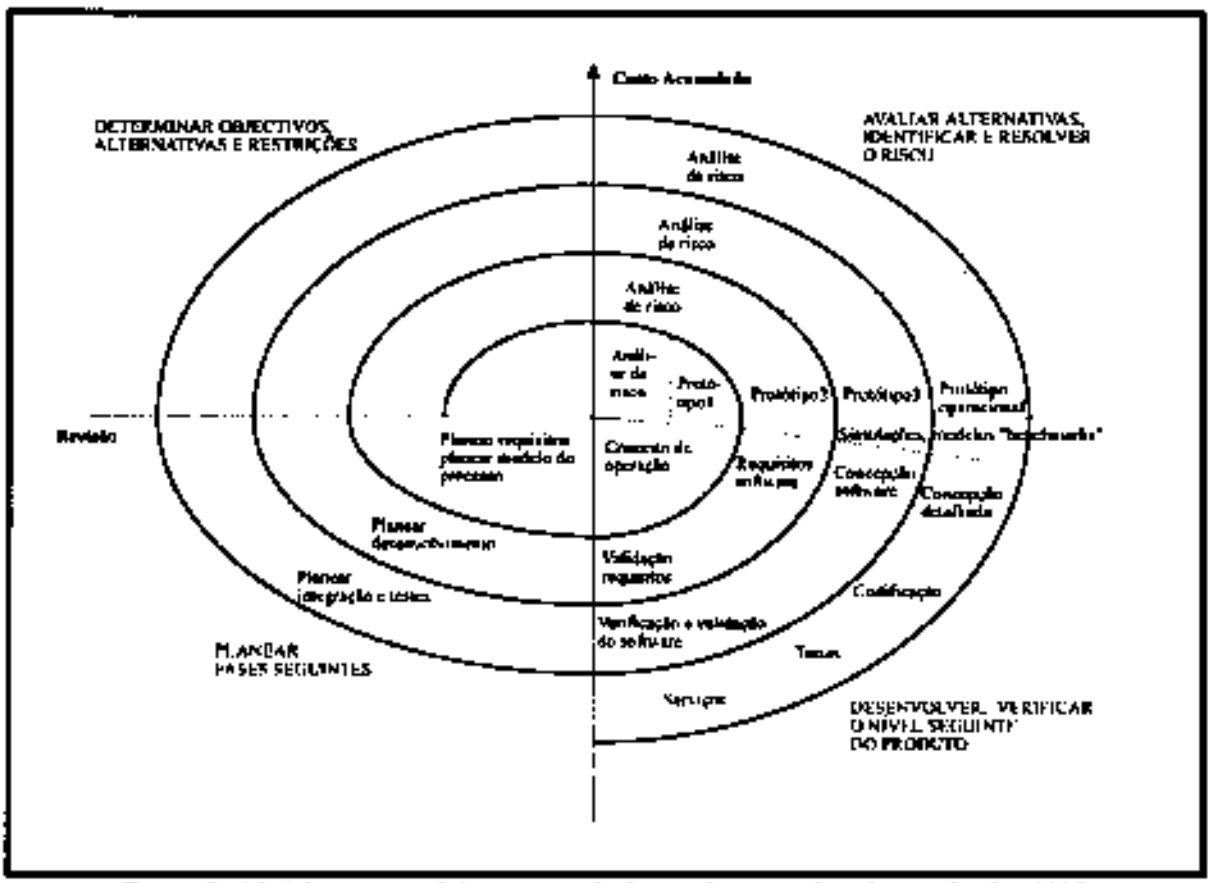


Figura 9 - Modelo em espiral do processo de desenvolvimento de software (Boehm 1987)

A dimensão radial da figura 9 represema o custo acumulado enquanto a dimensão angular representa o progresso realizado para completar cada ciclo da espiral.

A evolução do software faz-se através de vários ciclos e cada ciclo é constituido pela mesma sequência de etapas (Boehm 1987):

1. Determinar objectivos (performance, funcionalidade, etc), alternativas (reutilizar, comprar, desenvolver) e restrições (custo, escalonamento, interfaces, etc);

2. Análise do risco, pode envolver a prototipagem, simulação, modelos analíticos ou outras técnicas de resolução do risco;

3. Desenvolvimento do próximo nivel do produto;

4. Avaliação pelo cliente.

Considera-se este modelo do processo a abordagem mais realista ao desenvolvimento de software, embora na prática não seja possível representar com exactidão a sequência de tarefas realmente executadas (Kellner e Humphrey 1989). O ponto fraco deste modelo prende-se com a dificuldade de representar adequadamente os aspectos comportamentais do processo (Humphrey 1989)

2.4.6. Abordagem orientada aos objectos

Não existe um modelo do processo de desenvolvimento de software orientado aos objectos[13] universalmente aceite (Henderson-Sellers e Edwards 1990), (Rajlich 1985) Contudo vários autores concordam que estas técnicas representam uma abordagem radicalmente diferente ao processo de desenvolvimento de software, isto é, o desenvolvimento é realizado através da utilização de componentes de software. Uma aplicação é construida e não programada (Nierstrasz, Gibbs e Tsichritzis 1992) , Meyer (1990), Cox (1990). Embora o sonho de uma indústria de software baseada em componentes não seja novo (ver por exemplo, McIlroy 1969) só actualmente parece ser possivel a concretização deste sonho, através da massificação da tecnologia orientada aos objectos. Esta tecnologia apresenta uma base bastante boa para a reutilização de construções de software (Meyer 1990).

Para Nerson (1992) as tecnologias orientadas aos objectos permitem gerir a complexidade através de algumas ideias chave:

- As arquitecturas orientadas aos objectos são descentralizadas;
- A classificação faz parte da estrutura do sistema,
- As mesmas ideias e conceitos são manipulados desde a fase de requisitos até à implementação.

Nerson (1992) propõe um modelo orientado aos objectos do processo de desenvolvimento de software (fig 10) que segundo ele reflecte convenientemente um esquema de desenvolvimento em que uma base de conhecimento representada por uma biblioteca de

[13] Nierstrasz, Gibbs e Tsichritzis (1992) e Meyer (1990) preferem a denominação de modelo baseada na reutilização de componentes.

componentes reutilizáveis, tem um forte impacto nas diferentes fases de produção do software. A reutilização de classes e a generalização são um processo iterativo que influencia a análise, através da reutilização de estruturas (Coad 1992) e a concepção, através da reutilização das classificações (Meyer 1990).

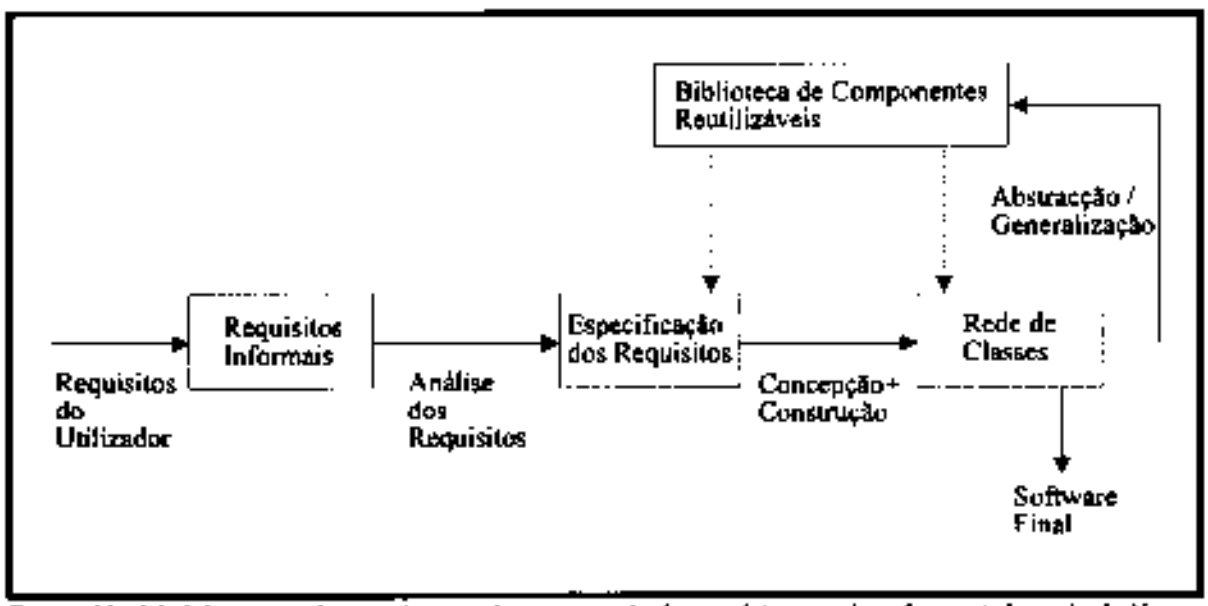


Figura 10 - Modelo orientado aos objectos do processo de desenvolvimento de software (adaptado de Nerson [1992])

Esta abordagem orientada à reutilização de componentes apresenta ainda algumas dificuldades (Nierstrasz, Gibbs e Tsichritzis 1992), de *ordem tecnológica*:

1. Actualmente as linguagens orientadas aos objectos não permitem um suporte completo ao desenvolvimento de software orientado aos componentes;
2. As ferramentas de desenvolvimento de software tendem a enfatizar a programação e "*debugging*" em vez da reutilização de componentes;

e de *ordem metodológica e cultural*:

3. A estrutura de concepção do problema, isto é, como fazer a abstracção do domínio de conhecimento necessário à reutilização;
4. Problemas de reutilização em larga escala, ou seja, como é possível obter um retorno satisfatório, do investimento de capital necessário à criação de componentes reutilizáveis de software.

2.5. Papel das ferramentas CASE no desenvolvimento de software

No comexto da engenharia de software, apresentou-se a evolução, as perspectivas e os niveis dos diversos modelos do processo de desenvolvimento de software. Ainda neste contexto é importante definir o papel das ferramentas CASE (*Computer-Aided Software Engineering*).

Pode-se definir ferramentas CASE, num sentido amplo, como ferramentas de software que fornecem assistência automatizada ao desenvolvimento de software e às suas actividades de manutenção e gestão do projecto (Vessey, Jarvenpaa e Tractinsky 1992). Segundo Forte e Norman (1992) as ferramentas CASE tiveram sucesso na automatização de muitas tarefas de rotina do desenvolvimento de software, na execução automática de algumas transformações entre representações, na introdução de métodos formais e semiformais no desenvolvimento de software e no aperfeiçoamento do trabalho cooperativo, demonstraram a capacidade de aumentar a produtividade e prevenir certos tipos de defeitos no software.

A evolução destas ferramentas foi significativamente influenciada pelas características das aplicações desenvolvidas em cada época e pelos métodos existentes para a sua construção (Norman e Chen 1992). Apresenta-se em seguida uma breve descrição da evolução das ferramentas CASE, adaptada de Norman e Chen (1992).

Em meados da década de 70, surgiu a primeira geração de produtos CASE (*upper* CASE), geralmente suportados por *mainframes* e baseados em texto. Estas ferramentas ajudaram a difusão dos métodos estruturados e salientaram a necessidade de ferramentas automáticas, para armazenar e analisar toda a informação gerada no desenvolvimento de software, pelos métodos estruturados.

A segunda geração de ferramentas CASE, baseada em ambientes gráficos, surgiu no início da década de 80. Estes produtos suportavam principalmente os métodos estruturados e os seus diversos diagramas, incorporando melhores funções de análise que permitiam a verificação das regras dos métodos. A informação armazenada no dicionário do projecto apenas podia ser integrada com ferramentas da mesma "*software house*" e os dados eram partilhados apenas dentro do mesmo projecto

Nos finais dos anos 80, surgiram os produtos CASE baseados num repositório central, que oferecem suporte à informação no nível organizacional e no nível do projecto. Normalmente incorporam ferramentas para o planeamento, análise, concepção, programação, testes e manutenção, mas suportam apenas um determinado método, pelo que tendem a ser utilizadas num só tipo de aplicações

Segundo Norman e Chen (1992), a década de 90 assistirá à construção de ambientes CASE integrados, como resposta à crescente procura dum desenvolvimento mais rápido, com melhor qualidade e integrado, de sistemas cada vez mais complexos. Para a construção e utilização de ferramentas CASE integradas considera-se necessário definir uma estrutura técnica e uma estrutura organizacional que suportem uma estratégia coerente de desenvolvimento (Chen e Norman 1992).

Chen e Norman (1992), propõem a adopção do modelo de referência (fig.11) do NIST/ECMA[14] (1991) para descrever os aspectos técnicos dum ambiente CASE integrado.

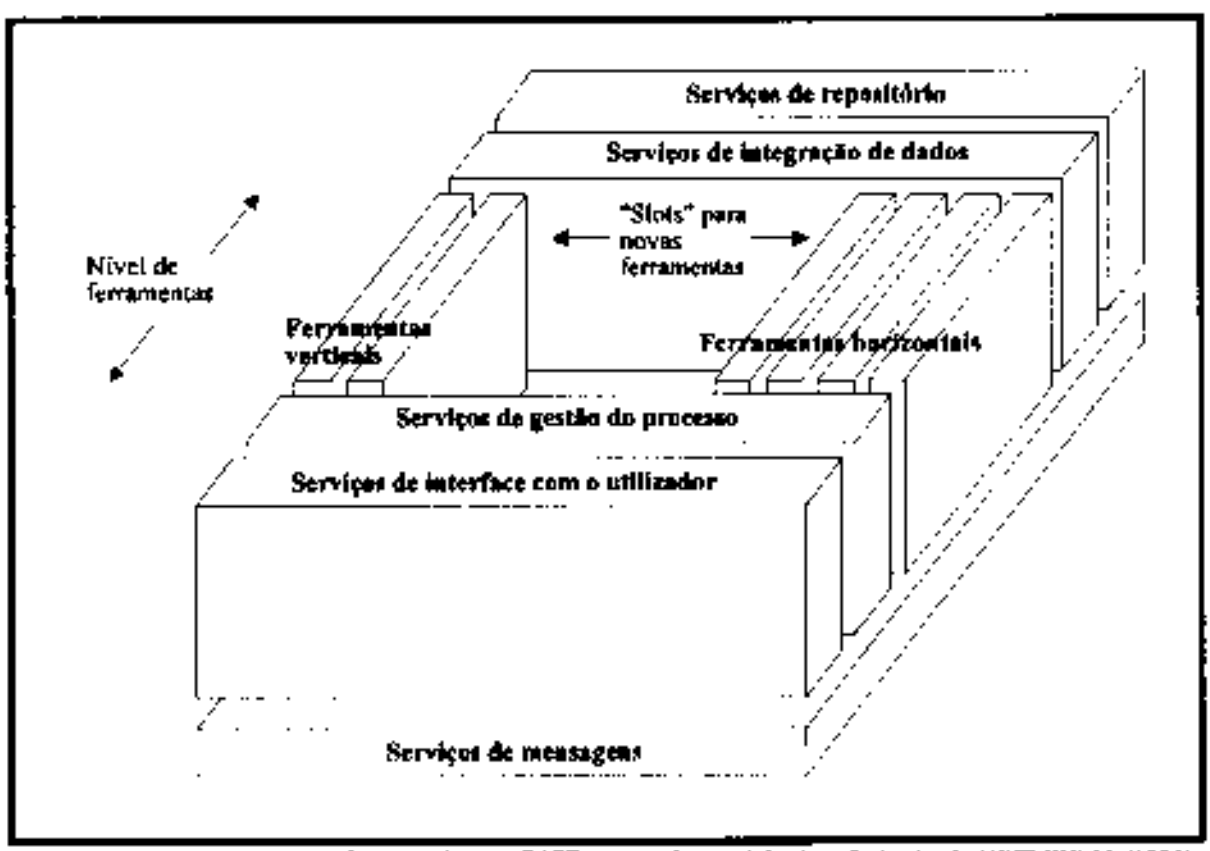


Figura 11 - Estrutura técnica de um ambiente CASE integrado, modelo de referência do NIST/ECMA (1991)

[14] *National Institute of Standards and Technology* e *European Computer Manufacturers Association*

Neste modelo são definidos serviços para permitir três formas de integração:

- Integração de dados, que é suportada pelos serviços de repositório e de integração de dados ;
- Integração de controlo, suportada pelos serviços de gestão do processo e serviços de mensagens;
- Integração de apresentação, suportada pelos serviços de interface com o utilizador.

O modelo prevê também a integração vertical e horizontal, no nível das ferramentas.

- Integração vertical, assegura a consistência da informação gerada nas diversas fases do processo de desenvolvimento;
- Integração horizontal, assegura a consistência da informação gerada pela utilização de diversos métodos numa mesma fase do processo de desenvolvimento.

Para Chen e Norman (1992) a estrutura organizacional (fig.12) permite colocar as ferramentas CASE integradas no contexto do desenvolvimento e gestão. No lado esquerdo da figura 12, agruparam-se as ferramentas e serviços do ambiente em três níveis. Os componentes em cada nível suportam as actividades correspondentes do lado direito.

A estrutura organizacional que completa a estrutura técnica apresentada deve orientar a evolução e desenvolvimento futuro dos ambientes CASE integrados e ajudar os utilizadores dos produtos CASE a seleccionar e configurar as ferramentas num ambiente CASE integrado (Chen e Norman 1992).

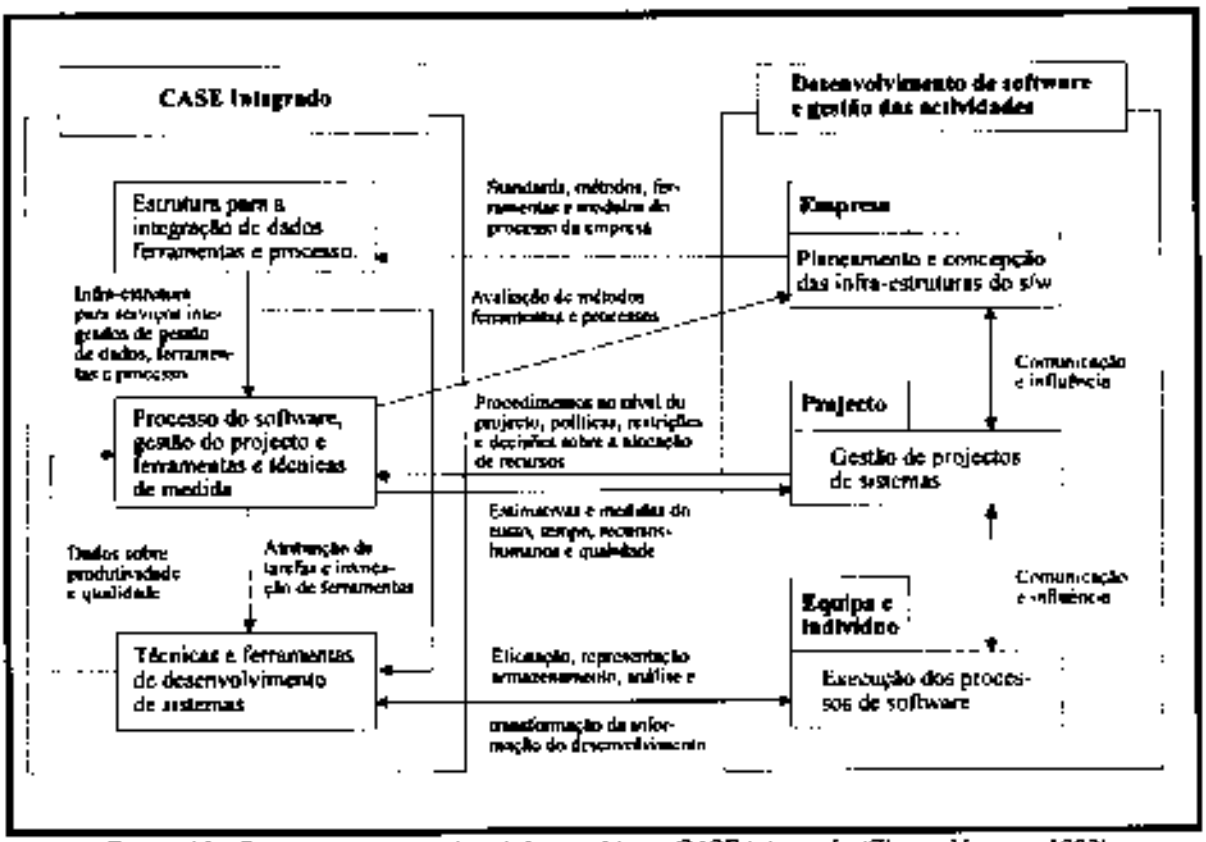


Figura 12 - Estrutura organizacional dum ambiente CASE integrado (Chen e Norman 1992)

Considera-se que os ambientes CASE integrados do futuro devem permitir o desenvolvimento através da combinação das melhores características das diferentes tecnologias emergentes (bases de dados, hipermédia, cliente / servidor, etc), devem suportar ferramentas especificas para o desenvolvimento cooperativo, devem prever a reutilização de componentes nas diversas fases do desenvolvimento, devem ajudar na transferência de tecnologia, devem adaptar-se a novos métodos e devem suportar a análise do comportamento organizacional (Chen e Norman 1992), (Norman e Chen 1992).

2.6. Gestão do processo de desenvolvimento de software

No estudo do processo de desenvolvimento de software, a perspectiva de gestão e controlo do próprio processo apresenta-se uma das áreas de maior importância para o reconhecimento da engenharia de software como uma verdadeira disciplina de engenharia (Forte e Norman 1992), (Basili e Musa 1991), (Cox 1990).

Nesta área, o trabalho de Humphrey "*Managing de Software Process*" (1989), é particularmente interessante. A equipa do Instituto de Engenharia de Software (SEI) liderada por Humphrey, desenvolveu um método de avaliação do processo, baseado no modelo de níveis de maturidade do processo e em simultâneo apresentou o projecto de avaliação das capacidades do software (SCE[15]) patrocinado pelo Departamento de Defesa dos Estados Unidos.

O método de avaliação do processo tem como objectivo ajudar as organizações a melhorar as suas próprias capacidades de desenvolver software, enquanto o método de avaliação das capacidades do software pretende ajudar os clientes a avaliar os seus fornecedores de software. Ambos os métodos utilizam como base de trabalho a análise dos níveis de maturidade do processo (fig.13), isto é avaliam e desenvolvem recomendações sobre o modelo do processo de desenvolvimento de software, numa perspectiva da sua gestão (Humphrey e Curtis 1991). Estes métodos têm uma grande aceitação nos Estados Unidos embora uma parte deste êxito se deva à adopção do SCE pelas entidades governamentais (Bollinger e McGowan 1991).

Humphrey (1989), apresenta um conjunto importante de conceitos relacionados com a gestão do processo de desenvolvimento de software. Segundo ele, a gestão do processo de software tem como objectivos produzir produtos de acordo com o plano e simultaneamente aperfeiçoar a capacidade da organização para produzir melhores produtos. Por outro lado, um processo de software para ser efectivo, tem que ser previsível, isto é, o comprometimento em relação à estimativa de custo e ao escalonamento deve ser cumprido e o produto final deve satisfazer as exigências de qualidade do utilizador. Este princípio, conhecido por controlo estatístico de um processo, foi aplicado com sucesso em diversos processos de fabricação de produtos e Deming (1982) apresenta-o afirmando que "um processo diz-se estável ou sob controlo estatístico se o seu desempenho futuro é previsível dentro de limites estatísticos estabelecidos". Uma melhoria contínua do processo só pode ser conseguida se o processo estiver sob controlo estatístico.

Segundo Humphrey (1989), uma organização para melhorar as suas capacidades de desenvolver software, tem que passar pelas seis fases apresentadas em seguida

1. Compreender o estado actual do seu processo de desenvolvimento;
2. Desenvolver uma perspectiva do processo desejado;

[15] *Software Capability Evaluation*.

3. Estabelecer uma lista de acções necessárias para o aperfeiçoamento do processo e definir a prioridade das acções;
4. Elaborar um plano para executar as acções requeridas,
5. Comprometer todos os recursos para a execução do plano;
6. Recomeçar na fase um.

Os niveis de maturidade (fig.13) são utilizados para estabelecer a posição da organização em relação a esta estrutura e assim identificar as áreas em que as acções de aperfeiçoamento irão com maior probabilidade produzir resultados (Humphrey 1988). Esta estrutura definida por Humphrey, baseia-se no trabalho de Crosby (1979) que definiu uma estrutura semelhante de maturidade da qualidade.

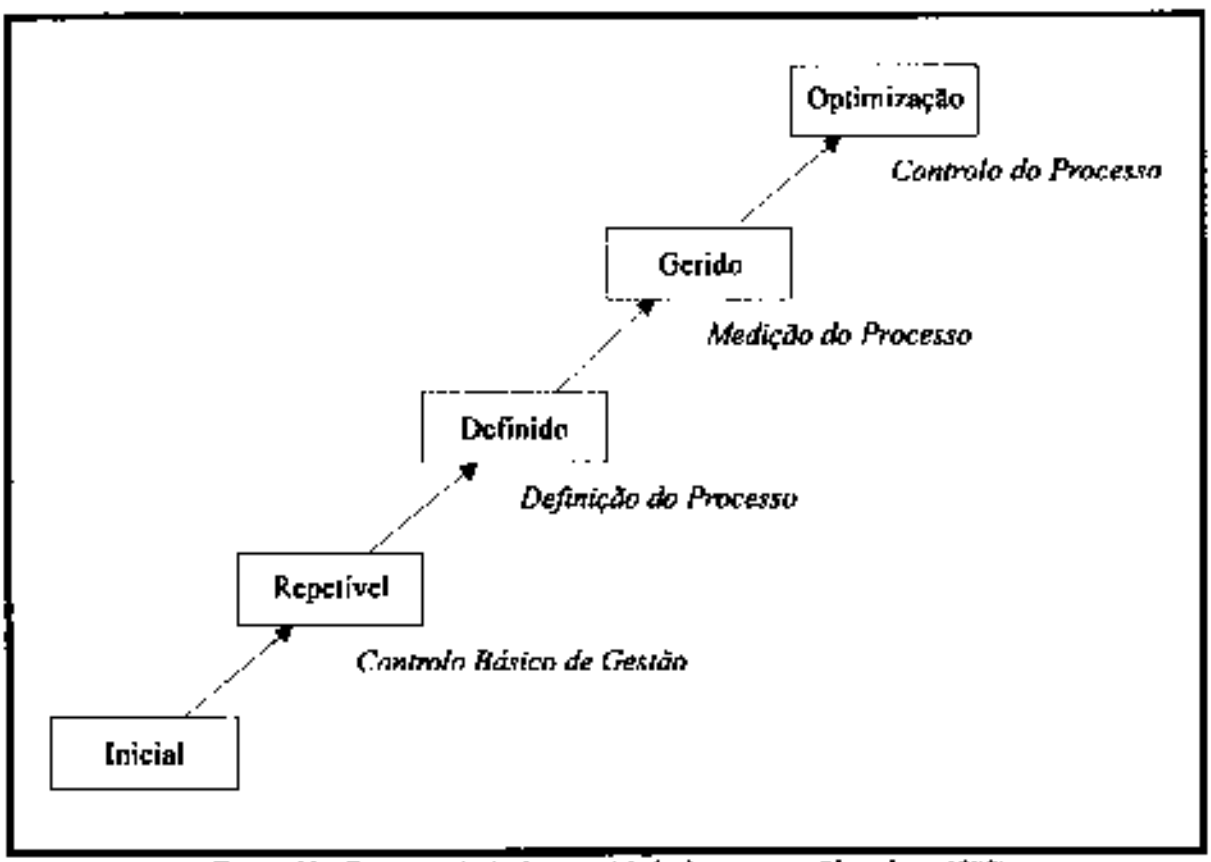


Figura 13 - Os cinco niveis de maturidade do processo (Humphrey 1989)

Sumariando, as caracteristicas e desafios criticos em cada nivel, são os seguintes (Humphrey, Snyder e Willis 1991):

- **Inicial** (processo caótico/ad-hoc), caracteriza-se pela ausência de procedimentos formais, estimativas de custo e planos dos projectos. Os

mecanismos de gestão não asseguram que os procedimentos são seguidos e as ferramentas não estão bem integradas. Os factores chave para o aperfeiçoamento são, a gestão e planeamento do projecto, a gestão de configurações e a segurança na qualidade do software.

- **Repetível** (processo intuitivo), caracteriza-se por possuir o controlo básico do projecto bem estabelecido mas o processo depende fortemente de alguns indivíduos. Este facto facilita o desenvolvimento de trabalhos similares, mas origina o aparecimento de grandes riscos quando a organização é sujeita a novos desafios. Por outro lado, apresenta-se frequentemente sem uma estrutura ordenada de aperfeiçoamento do processo. As acções criticas neste nível são a formação, a melhoria de procedimentos técnicos (revisões e testes) e alterar o centro de interesse para o processo, através da definição de normas e da criação da equipa do processo.

- **Definido** (processo qualitativo), caracteriza-se pela existência de um processo definido e institucionalizado e pela capacidade da equipa do processo liderar o aperfeiçoamento do processo de desenvolvimento de software. Os factores criticos são a medição e análise do processo e a definição de planos quantitativos sobre a qualidade.

- **Gerido** (processo quantitativo), caracteriza-se pela capacidade do processo ser medido, isto é, está estabelecido um conjunto mínimo de métricas sobre a qualidade e produtividade do processo Existe também uma base de dados centralizada sobre o processo e os recursos necessários para analisar e manter a informação. O aperfeiçoamento neste nível, baseia-se em mudanças tecnológicas e na alocação de recursos para a análise e prevenção de problemas.

- **Optimização**, caracteriza-se pela recolha automática de dados e sua utilização para identificar os elementos mais fracos do processo Utilização da evidência numérica para justificar a aplicação de tecnologia em determinadas tarefas criticas e a existência de uma análise rigorosa das causas dos defeitos e da sua prevenção Os desafios enfrentados são conseguir manter o processo no nível de optimização e incrementar a automatização do processo de desenvolvimento de software

A importância do método de avaliação do processo e do modelo da maturidade, centra-se na ajuda à identificação das capacidades actuais e dos pontos críticos do processo de desenvolvimento de software, isto é, uma identificação clara das prioridades de melhoramento. A estratégia usual das organizações é definir e melhorar o processo numa perspectiva técnica, esquecendo-se das práticas de gestão necessárias para o efectivo melhoramento do processo (Humphrey e Curtis 1991) A aplicação do método de avaliação do processo numa empresa[16], é descrito por Humphrey, Snyder e Willis (1991) que apresentam como benefícios principais, a melhoria da qualidade de trabalho, a melhoria da imagem da empresa e a diminuição do risco (melhoria no escalonamento e nas estimativas orçamentais[17]).

O método de avaliação do processo e o modelo da maturidade, apresentam algumas dificuldades Segundo Bollinger e McGowan (1991) os problemas surgem principalmente em três áreas: não apresenta uma perspectiva global e *top-down* de como o processo deve ser construído e optimizado; não reconhece o efeito dos diversos tipos de risco no processo de desenvolvimento de software; não entra em consideração sobre os aspectos tecnológicos no modelo da maturidade do processo. Sobre o método de avaliação da capacidade do software, Bollinger e McGowan (1991) reclamam a injustiça de classificar uma organização através de um questionário (85 perguntas, com respostas sim/não) e consideram mais justo a avaliação das provas dadas no desenvolvimento de software com qualidade comprovada

2.7. Perspectivas inerentes aos modelos do processo de desenvolvimento de software

A partir da descrição anterior dos vários modelos do processo torna-se claro que devem ser integradas diversas formas de informação para descrever adequadamente o processo de desenvolvimento de software. A equipa de desenvolvimento de software, pretende obter do modelo do processo a informação sobre o que será feito, quem está encarregado de fazer, quando e aonde será feito, como e porquê será desenvolvido e quem depende desse desenvolvimento (Curtis, Kellner e Over 1992). As respostas às perguntas anteriores, são obtidas através das diferentes perspectivas do modelo do processo de desenvolvimento de software Existem diversas opiniões sobre a classificação das diferentes perspectivas.

[16] *Hughes Aircraft*.
[17] Traduziu-se numa redução das despesas líquidas de cerca de 2 milhões de dólares.

Segundo Humphrey (1989), as perspectivas básicas dos modelos do processo são três:

- Perspectiva Organizacional, define as responsabilidades por cada actividade;
- Perspectiva do Estado, em que os estados representam as várias fases do processo (tarefas) ou as várias fases do produto;
- Perspectiva de Controlo, refere-se à medida e controlo do processo.

Moynihan (1993), apresenta cinco exemplos de perspectivas dos modelos do processo:

i. O desenvolvimento de software como uma actividade a ser planeada, controlada e organizada;
ii. O desenvolvimento de software como um processo criativo de concepção;
iii. O desenvolvimento de software como um processo colaborativo de resolução de problemas, envolvendo gestores, utilizadores e engenheiros de software;
iv. O desenvolvimento de software como um processo de construção, transformação e modificação de artefactos linguísticos abstractos;
v. O desenvolvimento de software como um processo que deve contribuir para a redução da incerteza dos clientes e analistas sobre os requisitos do software.

No âmbito desta dissertação interessa analisar o processo de desenvolvimento de software (em particular os sistemas de informação), na perspectiva das diversas representações possíveis, das transformações realizadas e na contribuição destes aspectos para a diminuição da incerteza sobre os requisitos do software (perspectiva *iv* e *v* propostas por Moynihan). Na secção seguinte, define-se um modelo do processo de desenvolvimento de software que cobre as duas perspectivas seleccionadas.

2.7.1. Modelo de representações/transformações do processo de desenvolvimento de software

Moynihan (1993) propõe um modelo do processo de desenvolvimento de software que combina duas das perspectivas por ele apresentadas: o desenvolvimento de software visto como uma manipulação de objectos linguísticos abstractos e como um processo de redução de incerteza. O desenvolvimento de software é apresentado como um processo evolutivo, em que a representação inicial, R_1, passa por uma sequência de representações intermédias (R_2,

R_3, ...), que culminam no software final, R_n. A "passagem" de R_i para R_{i-1}, representa-se por TS_i, isto é transformação$_i$ (fig.14).

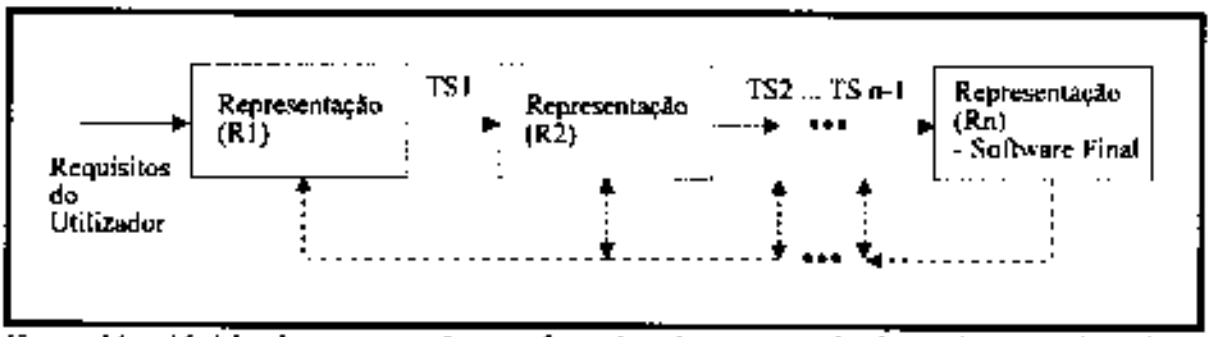


Figura 14 - Modelo de representações/transformações do processo de desenvolvimento de software (Moynihan 1993)

Pretende-se com esta abordagem, definir um conjunto de propriedades principais das representações, das transformações (TS) e dos clientes para obtermos uma estimativa qualitativa da satisfação do cliente e da qualidade final do software. As propriedades principais apresentadas por Moynihan são, em relação às representações: formalidade; executabilidade; complexidade de interpretação do conteúdo, complexidade de interpretação do comportamento, em relação às transformações: não similaridade, máximo rigor possivel na verificação; rigor planeado na verificação, e em relação ao cliente: compreensão do conteúdo; compreensão do comportamento; disponibilidade; incerteza nos requisitos.

Este modelo do processo de desenvolvimento apresenta algumas dificuldades que podem ser sumariadas em quatro áreas (Moynihan 1993):

1. O seu carácter qualitativo (é necessário definir qualidade em termos quantitativos);
2. Não captura a execução concorrente de tarefas;
3. Não reflecte o facto das representações serem muitas vezes fragmentadas (por exemplo, os módulos);
4. As propriedades seleccionadas não são consensuais.

2.7.2. Modelo adoptado do processo de desenvolvimento de software

No contexto desta dissertação, considera-se o modelo de representações / transformações, uma vez que analisa todo o processo de desenvolvimento de uma forma uniforme, isto é, um conjunto de representações em diversos sistemas linguísticos (linguagem

natural, especificação formal, linguagem gráfica ou uma linguagem de programação) e os processos de transformação que permitem a "passagem" entre duas representações. Adopta-se o modelo de Moynihan modificado (fig.15) como a representação mais apropriada do processo de desenvolvimento de software, isto é, a perspectiva mais adaptada aos propósitos deste trabalho.

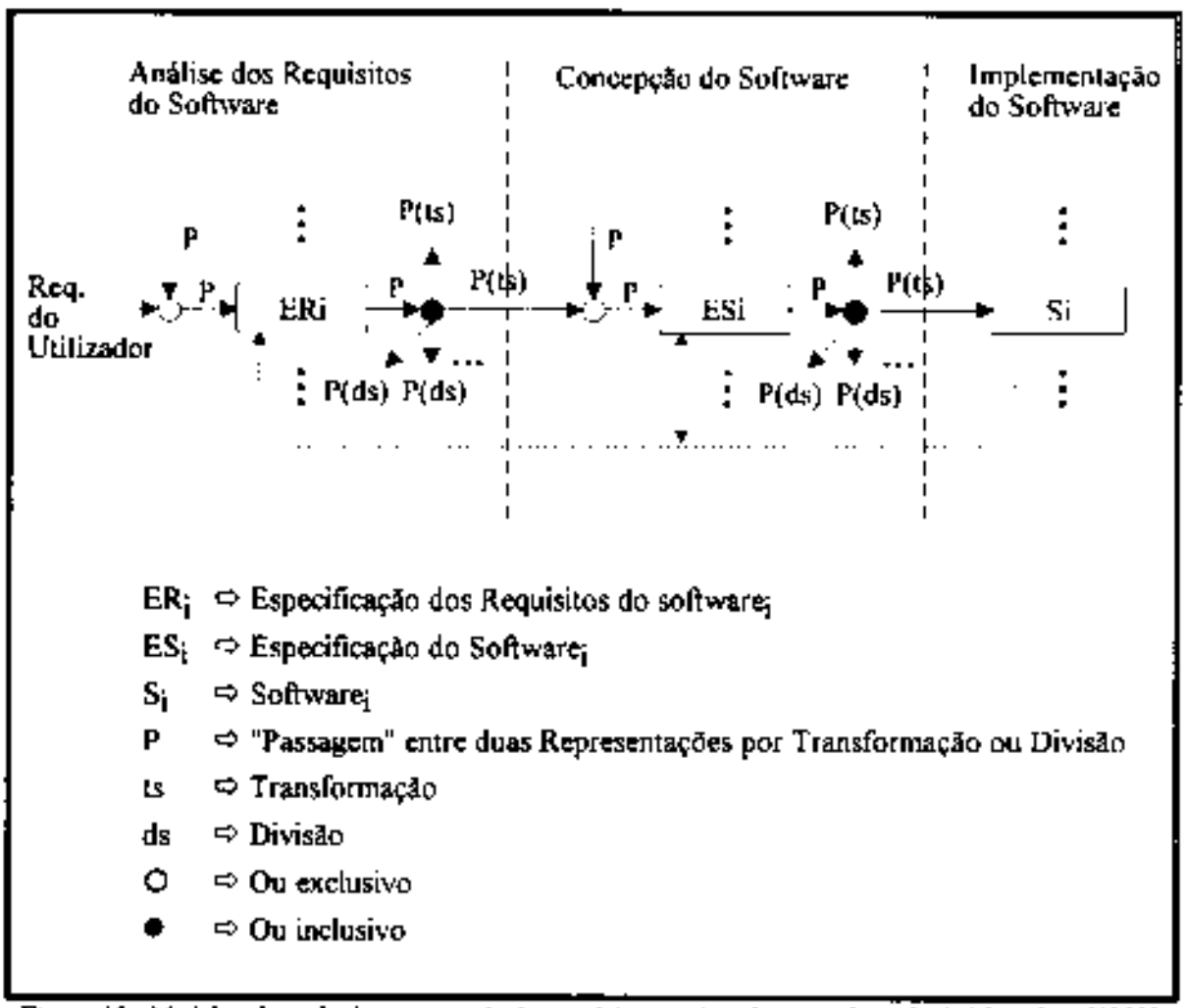


Figura 15 - Modelo adoptado do processo de desenvolvimento de software (adaptado de Moynihan [1993])

Pretende-se com este modelo "analisar" as diferentes representações existentes nas três primeiras fases do processo de desenvolvimento de software. Note-se que ER, ES e S são sempre representações. Por outro lado pretende-se flexibilizar o modelo do processo de modo a permitir a representação de tarefas concorrentes e permitir a divisão de uma representação. A Transformação - P(ts) - implica uma mudança de sistema linguístico, enquanto numa Divisão - P(ds) - o sistema linguístico se mantém verificando-se apenas uma fragmentação da representação.

Um possível exemplo da aplicação deste modelo do processo de desenvolvimento de software é apresentado na figura 16. O exemplo refere-se a um projecto hipotético de desenvolvimento de software e apresenta-se a seguir a sua descrição sumária.

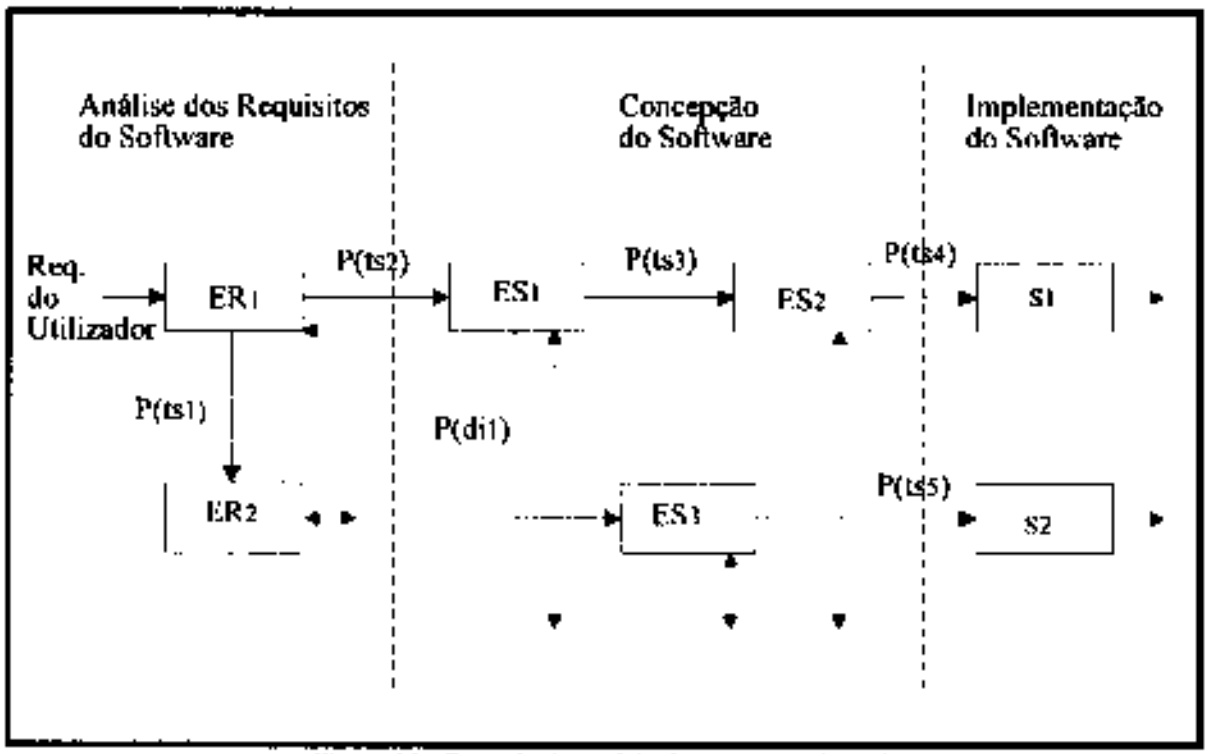


Figura 16 - Exemplo do modelo do processo adoptado

O projecto iniciou-se com a representação inicial (ER_1) dos requisitos do software, utilizando o método de desenvolvimento de software OMT (Rumbaugh et al. 1991). Através da transformação $P(ts_1)$, desenvolveu-se um protótipo (ER_2) utilizando a linguagem de especificação de requisitos, PAISLey (Zave e Shell 1986). O método concepção-estruturada (Page-Jones 1988), foi utilizado para desenvolver os diagramas de estrutura (ES_1), através da transformação $P(ts_2)$. Devido a considerações de fiabilidade e desempenho, utilizou-se o método formal VDM (Jones 1990), para desenvolver uma nova representação, ES_2 que formaliza detalhes da concepção de determinados módulos, executou-se a transformação $P(ts_3)$. Utilizando-se ainda a concepção-estruturada, detalhou-se parcialmente o software (ES_3) - a parte formalizada foi ignorada - através da divisão $P(ds_1)$. A implementação foi realizada em C (representação S_1), utilizando a transformação $P(ts_4)$ e em Cobol (Representação S_2), através da transformação $P(ts_5)$.

Ao analisarmos o modelo da figura 16 e a própria descrição do projecto, surgem algumas perguntas importantes:

1. Quais os principais tipos de representações que podem ser usados?

2. Que transformações existem entre que representações?

3. Que combinações entre representações e transformações serão mais convenientes?

As respostas para estas questões serão abordadas nos capítulos seguintes. Assim, na próxima secção, apresenta-se uma revisão de técnicas de representação, classificando-se cada técnica quanto à forma e conteúdo. Em seguida, apresenta-se uma revisão de transformações entre representações e finalmente no último capítulo, desenvolvem-se algumas conclusões sobre as representações e transformações mais convenientes no processo de desenvolvimento de software.

3. REVISÃO DE TÉCNICAS DE REPRESENTAÇÃO

No capítulo anterior, definiu-se a terminologia e conceitos principais do processo de desenvolvimento de software e apresentaram-se alguns dos modelos do processo mais usuais Caracterizaram-se também as diferentes perspectivas inerentes aos modelos do processo e seleccionou-se o modelo de representações / transformações, como o modelo mais adaptado para o âmbito deste trabalho. Este modelo visualiza o processo de desenvolvimento de software como um conjunto de representações suportadas por técnicas de representação e transformações entre representações.

Nesta secção apresenta-se uma revisão de técnicas de representação. Inicialmente define-se o enquadramento dos métodos de desenvolvimento de software no modelo de representações/transformações. Em seguida, descrevem-se diversas técnicas de especificação dos requisitos do software e técnicas de especificação do software, classificando-as quanto à forma e conteúdo. Finalmente, apresenta-se uma descrição global das linguagens de programação, classificando-as em relação ao paradigma de programação suportado.

3.1. Métodos de Desenvolvimento de software

Define-se método, numa perspectiva de engenharia, como uma abordagem detalhada para a resolução de um problema de engenharia (Thayer e Dorfman 1990).

Nesta perspectiva considera-se que o objectivo de um método é orientar os utilizadores no cumprimento de determinada tarefa especifica. É essencialmente um conjunto de regras e técnicas que orientam a evolução passo a passo. Um método de desenvolvimento

de software, dependendo do seu alcance, deve sugerir como as tarefas especificas, tais como, a análise dos requisitos do software, a concepção do software e a implementação, devem ser organizadas, abordadas e realizadas (Jones e Shaw 1990).

Karam e Casselman (1993, p.34) definem um método de desenvolvimento de software como "um sistema de procedimentos técnicos e convenções sobre notações, utilizados para a construção organizada de um sistema baseado em software".

Um método de desenvolvimento de software deve caracterizar-se por (Davies e Layzell 1993)

- um conjunto de técnicas para a análise, concepção e construção de um sistema;
- uma filosofia subjacente sobre o modo como o software deve ser desenvolvido,
- procedimentos para o planeamento e controlo do processo de desenvolvimento;
- um conjunto de ferramentas a serem utilizadas em conjunção com as técnicas propostas pelo método para a produção das diversas saidas (especificação dos requisitos, concepção preliminar, ...)

Na figura 17, apresenta-se os tópicos principais definidos por um método de desenvolvimento de software.

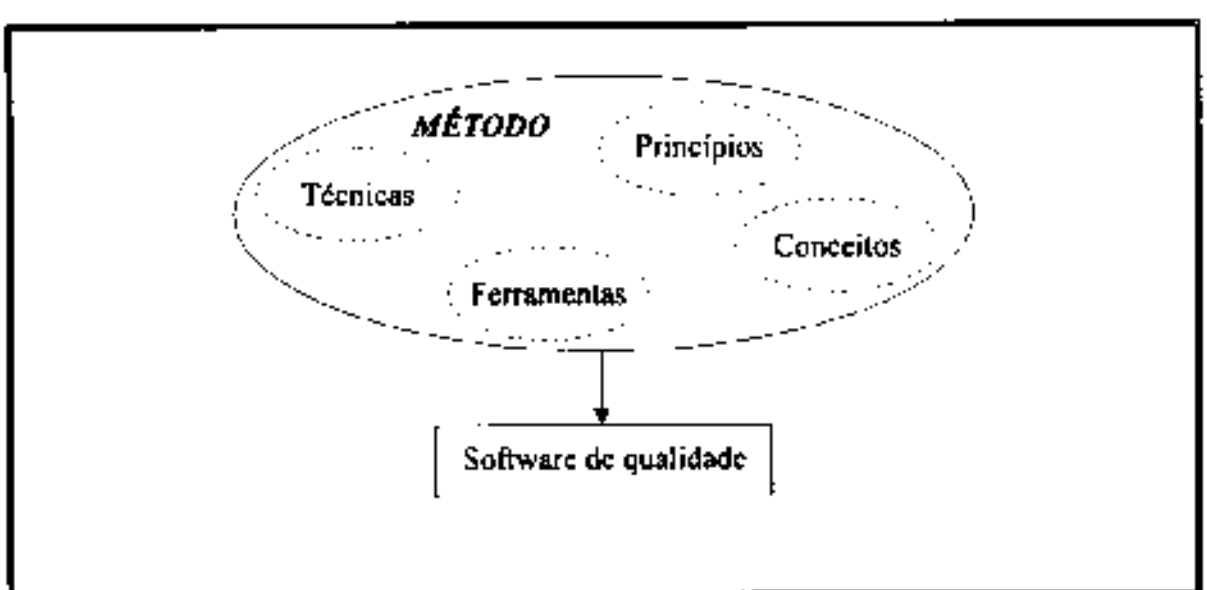


Figura 17 - Aspectos relevantes num método de desenvolvimento de software (baseado em Pfleeger [1991])

Hull e O'Donoghue (1993) classificam os métodos de desenvolvimento de software como apresentando técnicas para a análise dos requisitos, para a concepção do software ou endereçando ambas as etapas do processo de desenvolvimento de software. No âmbito desta dissertação, interessa analisar os métodos de desenvolvimento de software, na perspectiva das **técnicas de representação que sugerem**. Estas técnicas permitem o desenvolvimento de representações e são usados em uma ou mais etapas do processo de desenvolvimento de software.

3.2. Análise dos requisitos do software. Classificação das técnicas de representação.

Existem diversas opiniões sobre possíveis famílias de técnicas de representação, definidas pelos diferentes métodos de desenvolvimento de software.

Pfleeger (1991), divide as técnicas de representação dos requisitos do software, como:

- Descrevendo os aspectos estáticos do sistema;
- Descrevendo os aspectos dinâmicos do sistema;
- Orientadas aos objectos.

Por outro lado, Sommerville (1992), classifica os métodos de desenvolvimento de software como:

1. Métodos não formais,
    - Orientados às funções;
    - Orientados aos objectos.
2. Métodos formais,
    - Algébricos;
    - Orientados aos modelos.

Hodgson e Dunne (1990) e Olle et al. (1988), consideram que o desenvolvimento de software deve ser analisado segundo três perspectivas:

- Processos;
- Dados;
- Comportamento,

e as técnicas de representação dos requisitos do software, devem permitir modelar cada uma das perspectivas anteriores.

Para Ghezzi, Jazayeri e Mandrioli (1991) existem dois critérios alternativos para classificar as técnicas de representação dos requisitos do software,

1º Critério: Técnicas formais ou não formais.

2º Critério:

- Técnicas operacionais, descrevem o sistema através da descrição do comportamento desejado;
- Técnicas descritivas, expõem as propriedades desejadas de um sistema de um modo puramente declarativo.

Outra opinião é apresentada por Zave (1990), que refere a existência de três abordagens puras à especificação dos requisitos do software:

- Baseada na linguagem natural, semântica informal;
- Operacional, semântica executável;
- Matemática, semântica demonstrável.

Esta autora coloca cada uma destas abordagens nos vértices de um triângulo e classifica os métodos de desenvolvimento de software, pela sua aproximação às três abordagens puras.

Segundo Hull e O'Donoghue (1993), os métodos de especificação dos requisitos do software podem ser divididos em quatro classes:

- Funcionais;
- Formais;
- Textuais[18];
- Descritivos das operações do sistema[19].

Para Pressman (1992), as técnicas de representação dos requisitos do software, podem ser divididas em:

1. Orientadas aos fluxos de dados;
2. Orientadas ao comportamento do sistema;
3. Orientadas aos dados;
4. Orientadas aos objectos;
5. Representações formais.

[18] *Statement Oriented*.
[19] Transições de estados.

Finalmente, Yadav et al. (1988), referem que as diferentes técnicas de representação permitem modelar as diversas perspectivas de um sistema, nomeadamente:

- Actividades;
- Entradas / saídas;
- Definição de dados;
- Requisitos de processamento.

Considera-se que as técnicas de representação dos requisitos do software, devem ser analisadas quanto à **forma** e ao **conteúdo** (Ghezzi, Jazayeri e Mandrioli 1991).

Quanto à forma classificam-se em **técnicas formais** e **técnicas não formais** (Sommerville 1992), (Ghezzi, Jazayeri e Mandrioli 1991), (McDermid 1991) e (Fraser, Kumar e Vaishnavi 1991).

As técnicas formais baseiam-se em princípios matemáticos (McDermid 1991) e define-se uma técnica como formal, se apresentar uma notação[20], um universo de objectos[21] e regras precisas que definem quais os objectos que satisfazem a especificação (Wing 1990). As técnicas formais dividem-se em **algébricas** e **orientadas aos modelos** (Sommerville 1992), (McDermid 1991), (Wing 1988).

As técnicas formais orientadas aos modelos baseiam-se na definição de um modelo do sistema utilizando entidades matemáticas pré-definidas, tais como, conjuntos ou funções. As operações do sistema são especificadas definindo o modo como elas afectam o modelo global do sistema (Sommerville 1992). Em termos mais precisos, considera-se que a descrição da funcionalidade do sistema é apresentada com base num espaço de estado particular, em conjunto com uma colecção de operações e funções que actuam neste espaço. Estas operações e funções são expressos em termos duma colecção standard de tipos de dados básicos e outros construtores de tipos (McDermid 1991).

As técnicas algébricas baseiam-se no conceito de tipo abstracto de dados, ou mais precisamente, num modo de definição de tipos abstractos de dados a que se chama algébrico, ou axiomático. Esta abordagem baseia-se nos trabalhos de Guttag (1977). As classes de objectos ou tipos são especificados em termos dos relacionamentos entre as operações definidas nesse tipo (Sommerville 1992)

[20] Domínio sintático.
[21] Domínio semântico.

As técnicas não formais, caracterizam-se pela utilização de diagramas e gramáticas textuais semi-formais para a descrição dos requisitos do software (Fraser, Kumar e Vaishnavi 1991). Estas técnicas dividem-se em **técnicas diagramáticas e textuais**.

As técnicas diagramáticas, utilizam um conjunto de componentes visuais e regras precisas para o desenvolvimento dos modelos adequados. As técnicas textuais, baseiam-se numa linguagem textual, informal ou semi-formal, para descrever as propriedades do sistema.

Quanto ao conteúdo as técnicas de representação, classificam-se em: **orientadas às funções, orientadas aos dados, orientadas ao comportamento** e **orientadas aos objectos** (Pressman 1992) e (Pfleeger 1991).

As técnicas orientadas às funções descrevem as actividades ou processos do sistema, ajudando a compreender o modo particular como o sistema opera. O modelo funcional especifica o resultado das actividades, sem especificar como ou quando são realizadas (Rumbaugh et al. 1991). Considera-se que representa o sistema na perspectiva dos movimentos e transformações de dados (Downs, Clare e Coe 1992), ilustrando as funções que o sistema deve executar (Yourdon 1989).

As técnicas orientadas aos dados, apresentam o modo como os dados estão agrupados no sistema (Skidmore, Farmer e Mills 1992) e focam o seu interesse nos tipos de dados a armazenar, nas restrições aplicadas a estes dados e nos tipos de dados resultantes (Nijssen e Halpin 1989).

As técnicas orientadas ao comportamento, preocupam-se com as alterações sofridas pelo sistema e com a organização temporal dessas alterações (Pfleeger 1991). Controlo é o aspecto chave a modelar, isto é, o sistema é visto como uma sequência de operações que ocorrem como resposta a um estímulo externo, sem considerações sobre o que fazem as operações, sobre o que operam ou como são implementadas (Rumbaugh et al. 1991).

Nas técnicas orientadas aos objectos o sistema em análise é visto como uma colecção de objectos. A análise centra-se não só na identificação dos objectos, mas também nos serviços fornecidos pelos objectos, utilizando um modelo cliente-servidor de relacionamento entre objectos, em que os objectos interagem através de mensagens (Hendersen-Sellers e Edwards 1990). O objectivo principal é modelar a semântica do problema em termos de

objectos distintos, mas relacionados. Os objectos do domínio do problema representam coisas ou conceitos utilizados na descrição do problema (Monarchi e Puhr 1992).

A análise orientada aos objectos cobre também as três perspectivas referidas anteriormente, isto é, as funções, os dados e o comportamento. Contudo a terminologia utilizada e alguns conceitos são diferentes.

Assim, a orientação às funções denomina-se **serviços**. A perspectiva dos dados divide-se em **objectos, atributos e relacionamentos**. Considera-se que os relacionamentos incluem a agregação, generalização e qualquer outro tipo de associação[22]. Finalmente a perspectiva do comportamento (dinâmica) divide-se em **comunicação e temporização / controlo**. Verifica-se que o próprio objecto é o conceito unificador destas três perspectivas, ou seja, independentemente da visão do sistema que se analisa, o resultado desenvolve-se sempre em relação ao objecto.

Em resumo, obtém-se:

Técnicas de representação dos requisitos do software

FORMA

- Formais
  - Algébricas
  - Orientadas aos modelos
- Não formais
  - Diagramáticas
  - Textuais

CONTEÚDO

- *Orientadas aos objectos*
- Orientadas aos processos (*serviços*)
- Orientadas aos dados (*objectos, atributos e relacionamentos*)
- Orientadas ao comportamento (*comunicação e temporização / controlo*)

[22] Por exemplo o relacionamento *um elemento de* ou *uma perspectiva de*.

Na secção seguinte, apresentam-se algumas técnicas de representação dos requisitos do software, classificando-as quanto à forma e ao conteúdo. Tentou-se seleccionar as técnicas de representação pela sua maturidade, pela sua divulgação nos meios empresariais e académicos e pela sua representatividade em relação à classificação proposta.

3.2.1. Técnicas de representação formais

3.2.1.1. Técnicas de representação orientadas aos modelos

- **Linguagem Z**

A linguagem de especificação Z, atribui-se aos trabalhos de Abrial, Schuman e Meyer (1980), Sufrin (1983) e Morgan e Sufrin (1984). Neste capítulo apresenta-se as características mais importantes desta linguagem. Uma descrição completa desta linguagem pode encontrar-se em Spivey (1989) ou Hayes (1987).

A notação matemática da linguagem Z, baseia-se na matemática discreta (teoria de conjuntos e cálculo de predicados), que em conjunto com a linguagem de esquemas permite a produção de especificações estruturadas. A teoria de conjuntos permite que modelos abstractos do sistema sejam construídos, enquanto o cálculo de predicados fornece os conectivos lógicos para manipular os modelos. A linguagem de esquemas permite que os conceitos matemáticos sejam estruturados em entidades com significado, permitindo também o cálculo, para que um esquema possa ser manipulado de modo similar aos predicados (Norris 1986).

Nesta linguagem, utilizam-se essencialmente quatro tipos (McDermid 1991).

1. Tipos básicos ou conjuntos dados;
2. Tipos conjunto;
3. Tipos produto cartesiano;
4. Tipos esquema

Os tipos básicos ou são inteiros (Z) ou objectos atómicos obtidos de conjuntos dados. Dado um tipo básico ou composto podem-se construir conjuntos, contendo objectos desse tipo Utiliza-se normalmente o construtor P, 'conjunto de'. O produto cartesiano de dois

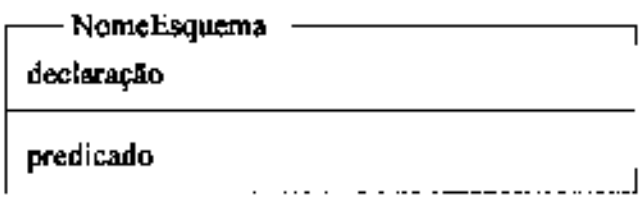
conjuntos A e B, define-se como o conjunto (designado por $A \times B$) de todos os pares (a,b) tal que a é um membro de A e b é um membro de B.

Considera-se que a linguagem de esquemas, fornece o suporte para a apresentação da especificação dos requisitos e para o desenvolvimento modular do software. Um esquema pode ser visto como uma construção que permite introduzir uma declaração (assinatura) e um predicado opcional. A declaração usualmente contém os nomes e tipos de uma ou mais variáveis e o predicado coloca restrições aos valores que podem ser adoptados pelas variáveis (Spivey 1989).

Um esquema pode apresentar-se, na forma horizontal,

NomeEsquema $\triangleq$ [declaração | predicado]

ou na forma vertical (normalmente preferida).



Um esquema é um mecanismo para designar estruturas matemáticas e utiliza-se para:

1. Descrever estados;
2. Descrever operações;
3. Definir tipos;
4. Escrever predicados;
5. Expor teoremas.

Esta ferramenta permite a estruturação da especificação e segundo McDermid (1991), apresenta as seguintes vantagens:

- Subordinação da complexidade;
- Mostrar as diferenças entre estruturas similares;
- Reutilização de partes existentes;
- Modificação simplificada de partes para a construção de novas partes.

Um dos pontos mais importantes na utilização da linguagem de esquemas refere-se à especificação das operações observadas no sistema. Uma operação deve observar-se em relação à alteração de estado, aos argumentos necessários e aos possíveis resultados (Sommerville 1992).

Na especificação do efeito de uma operação utilizam-se as variáveis de estado. A especificação baseada em modelos, baseia-se na ideia de expor os relacionamentos entre os valores dos componentes do estado, antes da operação ser iniciada e depois da operação estar completa. Em Z, por convenção, cada variável de estado aparece de uma forma decorada ou não decorada. Na forma decorada, o nome da variável de estado tem o simbolo ' como sufixo e designa o valor do estado final.

A convenção Ξ (xi), utiliza-se em operações em que o estado não é alterado pela operação e a convenção Δ (delta), utiliza-se em operações em que existe uma alteração do estado, isto é, uma ou mais variáveis de estado que sofrem alterações pela operação.

As operações requerem a passagem de argumentos. Em Z, por convenção os argumentos de uma operação são modelados por variáveis decoradas com o simbolo ?. Os argumentos são vistos como constantes e os seus valores não podem ser alterados. A produção de resultados é modelada por variáveis decoradas com o simbolo !.

Outro aspecto importante refere-se à possibilidade de combinar esquemas, utilizando os operadores lógicos de esquemas. Estes operadores fornecem um mecanismo flexível para combinar esquemas em unidades de especificação incrementalmente mais complexas (McDermid 1991). Os operadores lógicos de esquemas são:

Conjunção	Esquema1 ∧ Esquema2
Disjunção	Esquema1 ∨ Esquema2
Quantificador existêncial	∃ declaração • Esquema
Quantificador universal	∀ declaração • Esquema
Exclusão	Esquema \ (a_1,a_2,...)

Segundo Norris (1986), uma das caracteristicas principais dos métodos formais é a capacidade de provar que a especificação está de acordo com os requisitos do cliente. Este objectivo obtém-se expressando um requisito como um teorema e utilizando a especificação para provar que o teorema é verdadeiro. Em Z, um teorema é expresso em termos de

predicados e esquemas e utilizando-se o método de prova adequado demonstra-se que a especificação está de acordo com os requisitos.

O Z adapta-se melhor a problemas sequenciais orientados aos dados, embora possa ser utilizado também para especificar sistemas em tempo real ou distribuidos (Norris 1986).

Da descrição anterior, conclui-se que a técnica de representação Z, permite descrever a perspectiva funcional, a perspectiva dos dados e o comportamento de um determinado sistema (McDermid 1991).

- **Linguagem VDM**[23]

Nesta secção apresenta-se as caracteristicas principais desta linguagem Uma descrição completa do método poderá ser encontrada em Jones (1990) e a sua aplicação a diversos casos práticos encontra-se em Jones e Shaw (1990).

O método VDM foi originalmente utilizado como um mecanismo para definir com precisão a estrutura e significado de linguagens de programação Subsequentemente foi aperfeiçoado para um método de desenvolvimento de software de aplicação global e é utilizado numa grande variedade de domínios, nomeadamente, bases de dados, sistemas operativos e sistemas em tempo real. A notação de base utilizada designa-se por 'Meta-IV' (Ince 1989).

O VDM é uma abordagem orientada aos modelos, isto é, a descrição do sistema desenvolve-se a partir do desenvolvimento de modelos. Os constituintes dos modelos são objectos de dados, representando as entradas, saidas e o 'estado' interno do sistema e operações e funções que manipulam os dados. O VDM baseia-se então no reconhecimento que um tipo de dados é caracterizado pelas operações que podem ser executadas sobre os seus valores (Hull e O'Donoghue 1993).

O analista que utiliza o VDM, primeiro identifica os objectos de dados e operações que ocorrem no sistema a modelar e depois especifica-os (Ince 1989) utilizando um conjunto de tipos de dados pré-definidos, tais como, conjuntos, listas e mapeamentos (Wing 1990). Depois dos objectos de dados serem definidos o efeito das operações é clarificado.

[23] *Vienna Development Method.*

Considera-se uma operação como sendo um procedimento abstracto que, dado os argumentos com tipos apropriados e o acesso aos componentes do estado, pode produzir resultados e provocar alterações ao estado. As operações podem utilizar-se nos vários níveis de abstracção, capturando a semântica desde fragmentos no nível do programa até interacções complexas no nível do sistema. Segundo McDermid (1991) a especificação de uma operação possui três partes em VDM.

1. Assinatura, fornece informação sobre os argumentos e resultados da operação e apresenta, utilizando a palavra reservada ext (clausula externa), quais os componentes do estado que a operação vai ler (rd) ou ler/escrever (wr);

2. Pré-condição, é um predicado que regista quais as suposições que são feitas sobre os argumentos e sobre os componentes de estado acedidos pela operação. A pré-condição tem que ter como resultado, o valor lógico verdadeiro, para ser garantido que a operação termine.

3. Pós-condição, é um predicado que mostra a relação que tem que existir entre os valores de entrada, os valores de saída e os valores acedidos pela operação. Na pós-condição pode-se pretender referenciar o valor das variáveis de estado antes da invocação da operação e depois de estar completa. Quando nos referimos ao valor de uma variável de estado, antes da invocação da operação coloca-se um traço sobre o nome da variável, por exemplo, $\overline{\text{variável}}$.

O método standard para o desenvolvimento de programas a partir de especificações do VDM, envolve o refinamento dos objectos de dados. Em cada transformação as estruturas abstractas de dados são refinadas para estruturas de dados orientadas à implementação com as consequentes modificações na especificação das operações. Eventualmente as estruturas de dados e operações serão especificadas de um modo suficientemente concreto que permitem uma representação directa numa linguagem de programação. É então necessário mostrar que o código produzido satisfaz correctamente a última especificação abstracta (Hull e O'Donoghue 1993).

Se analisarmos esta técnica de representação quanto ao seu conteúdo, considera-se que permite descrever os dados, as funções e o comportamento de um sistema (Prehn 1987).

3.2.1.2. Técnicas de representação algébricas

Segundo McDermid (1991), a abordagem algébrica, define um tipo de dados a partir de um ou vários conjuntos de valores e de algumas operações sobre estes valores. A assinatura de um tipo de dados é constituida pelos seus tipos, os nomes das operações e a cardinalidade destas operações, isto é, os tipos do seu domínio e contra domínio.

Considera-se que um tipo abstracto de dados é uma classe de álgebras com a mesma assinatura e algumas propriedades especificas comuns. Um tipo de dados algébrico é a definição de um tipo abstracto de dados, através da assinatura e axiomas (fórmulas lógicas) que as álgebras da classe têm que satisfazer. Assim um tipo algébrico de dados é um par (Σ,E), em que Σ é a assinatura e E é um conjunto de axiomas_Σ (McDermid 1991).

Apresenta-se em seguida um resumo de duas técnicas de representação algébricas Larch (Guttag e Horning 1986) e PLUSS (Gaudel 1985).

- **Técnica de representação Larch**

A técnica de representação Larch combina uma componente axiomática, que especifica o comportamento dependente do estado e uma componente algébrica que especifica as propriedades dos dados independentes do estado.

A primeira parte de uma especificação em Larch, designa-se especificação do interface. Utiliza-se as clausulas *requires* e *ensures* para especificar respectivamente a pré e pós condição de cada operação. A clausula *modifies* apresenta os objectos cujo valor pode ser alterado como resultado da execução da operação.

A segunda parte de uma especificação Larch denomina-se *trait* e contém um conjunto de declarações de símbolos de função e um conjunto de equações que definem o significado dos símbolos de função (Wing 1990).

Considera-se que a técnica de representação Larch permite representar a perspectiva funcional, a perspectiva dos dados e o comportamento de um sistema.

- **Técnica de representação PLUSS[24]**

Esta linguagem fornece uma forma de estruturar especificações algébricas, isto é, qualquer especificação para a qual a semântica formal possa ser dada em termos de uma assinatura e uma classe de álgebras.

A assinatura (usualmente denominada Σ) define um conjunto de tipos, nomes de operações (funções) e um conjunto de nomes de predicados, por exemplo:

_ + _ : Inteiro * Inteiro → Inteiro
_ is empty : Bag

Define-se um termo como uma composição válida de variáveis, predicados e operações de Σ. Os axiomas são fórmulas de primeira ordem, construidas a partir de fórmulas atómicas (termos) e dos seguintes conectores lógicos. &, OR, ⇒ e iif.

Segundo Bidoit, Gaudel e Mauboussin (1989), a principal originalidade do PLUSS, reside na distinção entre os componentes da especificação completos e os componentes ainda em análise. Esta distinção obtém-se através da utilização das palavras chave *spec* e *draft*, respectivamente.

Outros aspectos importantes desta linguagem são:

- utilização do construtor *use* como forma de acrescentar incrementalmente novas características a uma especificação já existente;
- utilização de especificações genéricas (parametrização), através dos construtores: *proc* (definição da especificação parametrizada); *param* (definição dos parâmetros formais), e *as* (mecanismo de instanciação);
- utilização do construtor is defined when para definição das pré condições;
- utilização de frases completas como nomes de operações ou predicados para melhorar a legibilidade da especificação.

[24] *a Proposition of a Language Usable for Structured Specifications.*

A técnica de representação PLUSS, tal como o Larch, permite especificar a perspectiva dos dados, funcional e o comportamento do sistema em análise.

3.2.2. Técnicas de representação não formais

3.2.2.1. Técnicas de representação diagramáticas

As técnicas diagramáticas utilizam um conjunto de componentes visuais e regras precisas para o desenvolvimento dos modelos adequados. Verifica-se que a maioria das técnicas de representação mais divulgadas pertencem a esta classe (Yourdon 1989).

Na revisão das técnicas pertencentes a esta classe utiliza-se como modo de classificação a sua orientação principal: funções, dados, comportamento ou objectos.

3.2.2.1.1. Orientadas às funções

- **Diagrama de Fluxo de Dados**[25]

Os primeiros trabalhos sobre os diagramas de fluxo de dados (DFD), surgem ligados aos conceitos de "concepção estruturada" e devem-se a Stevens, Myers e Constantine (1974) e Yourdon e Constantine (1978) Contudo a ferramenta foi popularizada por DeMarco (1978) que introduziu e caracterizou os componentes principais dos DFDs Surgiram depois, diversas variações desta técnica, por exemplo Page-Jones (1988) ou Gane e Sarson (1983) Uma descrição completa da notação encontra-se em Yourdon (1989) ou Skidmore, Farmer e Mills (1992).

Um diagrama de fluxo de dados é uma técnica gráfica que descreve o fluxo de informação e as transformações que são aplicadas conforme os dados fluem da entrada para a saída. Esta técnica utiliza-se para representar um sistema ou software em qualquer nível de abstracção. De facto, os DFDs podem ser divididos em níveis que representam incrementos no detalhe das funções e dos fluxos de informação (Pressman 1992).

[25] *Data Flow Diagram*.

Segundo Martin e McClure (1991) um DFD é uma representação em rede de um sistema que mostra os processos e os fluxos de dados entre eles. Um DFD, constrói-se a partir de quatro componentes básicos: o processo, o fluxo de dados, o depósito de dados e as entidades externas (terminadores). Na figura 18, apresenta-se duas notações para os DFDs.

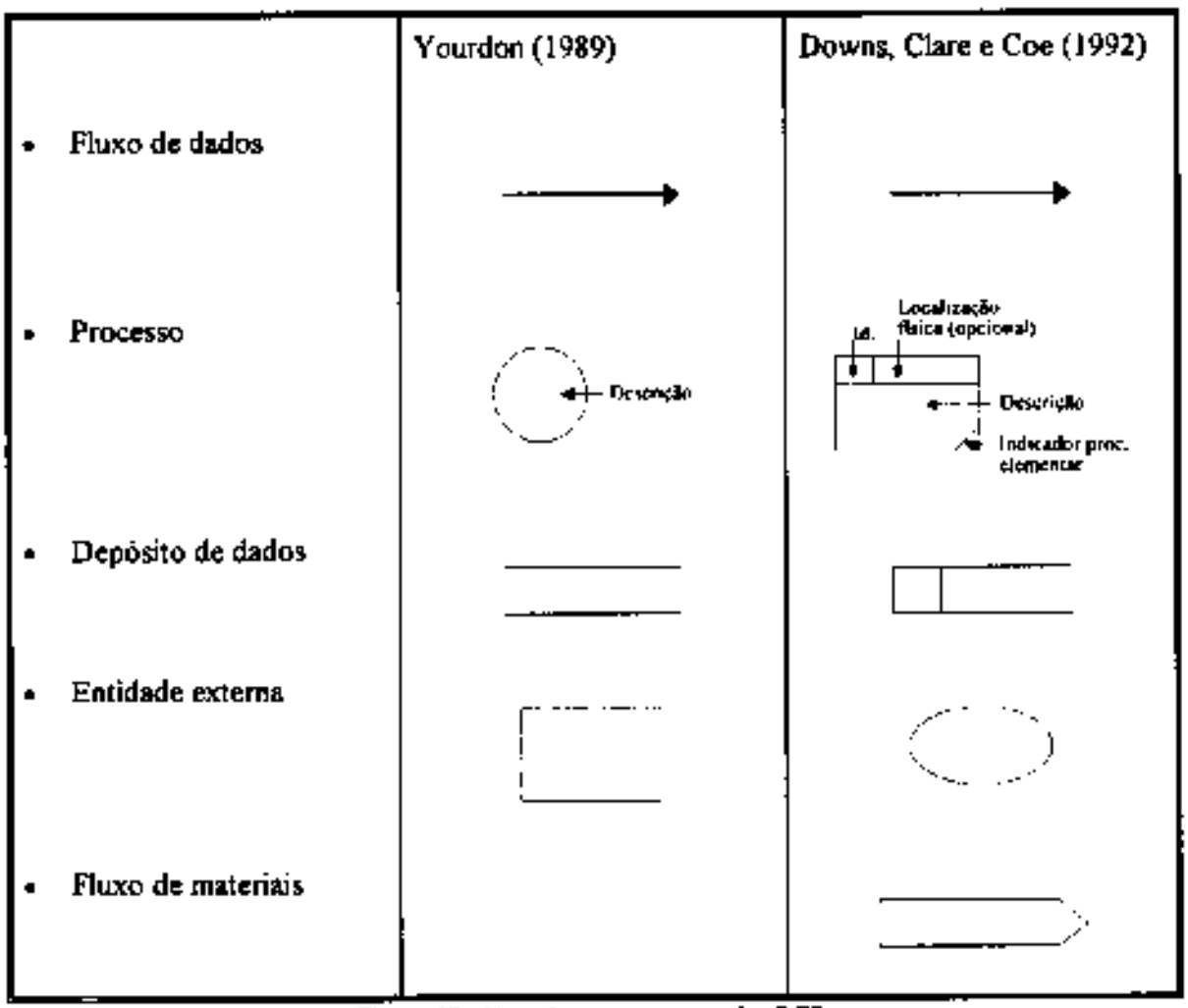


Figura 18 - Notações mais comuns dos DFDs

Os processos são transformações, alterando os fluxos de dados de entrada para fluxos de dados de saída. Todos os processos são numerados e o seu nome deve ser significativo, isto é, deve sugerir a operação executada pelo processo (Skidmore, Farmer e Mills 1992).

Um fluxo de dados permite representar o movimento de informação entre os diversos objectos do sistema. O sentido do fluxo de dados é indicado por uma seta e associa-se a cada fluxo um nome indicativo do seu conteúdo (Yourdon 1989).

O depósito de dados utiliza-se para modelar uma colecção de informação estática. O depósito de dados pode ter um suporte manual ou automático, contendo informação permanente ou temporária (Martin e McClure 1991).

A entidade externa mostra a origem ou destino dos dados utilizados pelo sistema. É portanto um subsistema, por exemplo, um departamento, uma pessoa ou uma organização que interage com o sistema em análise. Para Yourdon (1989), existem três aspectos sobre as entidades externas que devem ser considerados:

- As entidades externas estão fora da fronteira de análise, por isso, o seu conteúdo e modo de funcionamento não podem ser alterados;
- Os fluxos entre as entidades externas e os processos representam o interface entre o sistema e o mundo externo;
- Os relacionamentos entre entidades externas não devem ser apresentados, uma vez que não fazem parte do sistema em estudo.

O diagrama de fluxo de dados utiliza um modo simples e consistente para representar os níveis sucessivos de modelação necessários para apresentar uma hierarquia. Cada processo é expandido num DFD, de nível mais baixo, até que os processos sejam pouco complexos e possam ser explicados por um texto descritivo (especificação do processo). Deste modo, é possível apresentar uma série de DFDs, com um nível de detalhe crescente e apropriados para tarefas e equipas diferentes (Skidmore, Farmer e Mills 1992).

Esta notação básica utilizada no desenvolvimento dos DFDs, necessita do apoio de duas ferramentas para descrever os objectos intervenientes. Estas ferramentas adicionais são (Yourdon 1989):

- Dicionário de dados,
- Especificação dos processos

Segundo Yourdon (1989, p.236), "o dicionário de dados é uma lista organizada de todos os elementos de dados, pertencentes aos DFDs. A notação deve ser precisa e rigorosa para que os utilizadores e analistas tenham uma compreensão comum, das entradas, saídas, depósitos de componentes e até cálculos intermédios"

A especificação dos processos é utilizada para descrever todos os processos do nível final de refinamento e define o que deve ser feito para transformar os fluxos de entrada em fluxos de saída. Existem diversas notações que podem ser utilizadas para produzir uma

especificação de processos: tabelas de decisão, linguagem estruturada, árvores de decisão e outras (Davis 1990).

A maioria dos analistas aconselha a linguagem estruturada[26] (Yourdon 1989), (Gane e Sarson 1983) e (Martin e McClure 1991), que se caracteriza pela utilização de um subconjunto da linguagem natural com algumas restrições quanto ao tipo de frases que podem ser utilizadas e quanto ao modo como são construidas.

Em conclusão pode-se considerar que os DFDs apresentam três caracteristicas significativas (Skidmore, Farmer e Mills 1992):

- carácter gráfico, facilmente perceptivel pelos intervenientes;
- consistente e definição fácil da hierarquia;
- acessivel aos utilizadores que normalmente definem o sistema em termos de processos e transformações.

A importância desta técnica verifica-se pela sua utilização em diversos métodos de desenvolvimento de Software, tais como, SSADM (Downs, Clare e Coe 1992), OMT (Rumbaugh et al. 1991), SA/SD (Yourdon 1989), IE (Martin 1990) e outros.

- **Diagrama de especificação do sistema** [27]

A apresentação dos diagramas de especificação do sistema (DES) deve-se a Jackson (1983) e esta técnica de representação dos requisitos do software utiliza-se exclusivamente no método de desenvolvimento de software JSD (*Jackson System Development*).

O diagrama de especificação do sistema é uma técnica de representação que permite modelar o sistema como uma rede de processos interligados, fornecendo uma perspectiva de alto nível do sistema (Sutcliffe 1988). Cada processo pode conter os seus dados locais e os processos comunicam entre si através da leitura e escrita de mensagens e acessos só de leitura aos dados uns dos outros (Cameron 1986).

Segundo Davies e Layzell (1993) o diagrama de especificação do sistema[28] permite a criação de um mapa lógico do sistema, mostrando como os processos individuais comunicam

[26] Ver secção 3.2.2.2.
[27] *System Specification Diagram* (SSD).
[28] Também designado diagrama de rede (Davies e Layzell 1993)

entre eles e com o mundo real. Assume-se que todos os processos são concorrentes, isto é, são capazes de correr independentemente, parando apenas quando necessitam de dados de entrada não disponíveis.

Os diagramas de especificação do sistema são construidos a partir de três componentes:

- Processo;
- Fluxo de dados[29];
- Vector de estado

Os processos são representados por rectângulos e podem ser processos do modelo ou processos função. Esta distinção enfatiza a distinção clara entre os componentes mais importantes e estáveis do sistema (processos do modelo) e os componentes acrescentados com o progresso da modelação (processos função), por exemplo, validação de entradas, formatação de relatórios, etc (Sutcliffe 1988).

Um fluxo de dados representa-se por uma circunferência (fig 19) e corresponde a um canal de mensagens com um *buffer* organizado numa base FIFO (*first in / first out*). Os fluxos de dados provocam uma comunicação controlada entre os processos, já que por definição, se o processo que escreve não produzir nenhuma mensagem, o processo de leitura é obrigado a esperar até à chegada de uma nova mensagem.

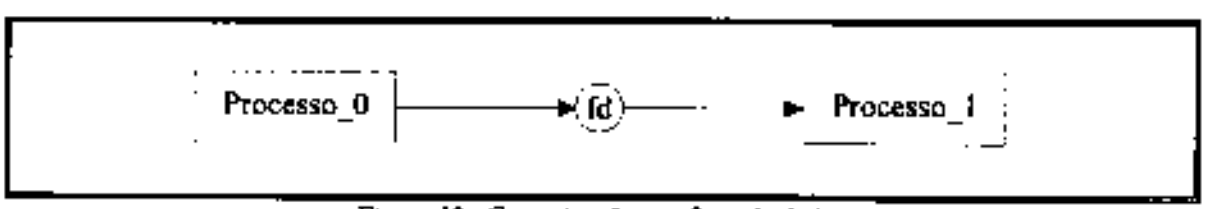


Figura 19 - Comunicação por fluxo de dados.

Características mais importantes dos fluxos de dados (Davies e Layzell 1993):

- Os dados têm que ser processados sequencialmente,
- Comunicam através de um *buffer* não limitado;
- A falta de dados provoca a paragem do processo que lê;
- São designadas por conexões fortemente acopladas;
- São a forma principal de ligação ao mundo real.

[29] *Datastream*.

O vectores de estado são representados por losangos no diagrama de especificação do sistema (fig.20). Os vectores de estado não são mensagens, mas sim dados que pertencem e descrevem um processo particular. A ligação por vector de estado é como uma instrução de inspecção e não provoca uma comunicação controlada entre processos (Sutcliffe 1988).

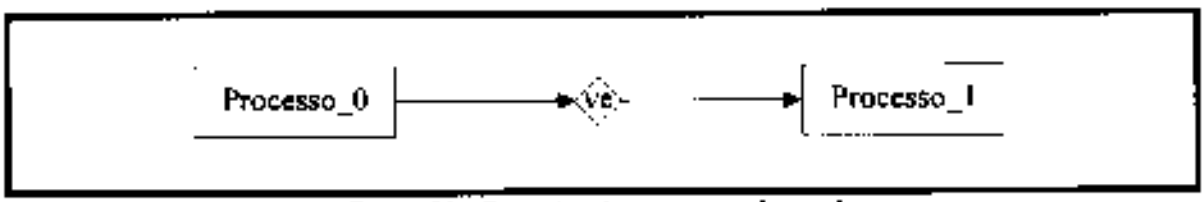


Figura 20 - Comunicação por vector de estado.

Caracteristicas mais importantes da comunicação por vector de estado (Davies e Layzell 1993):

- Representam o estado interno do processo;
- Podem ser lidos sem alterações desde a última inspecção,
- Existe sempre alguma coisa para ser lida;
- Não exercem controlo no processo que inspecciona.

Em síntese, considera-se que a comunicação por fluxo de dados é utlizada primariamente para passagem de mensagens de eventos através do sistema, enquanto a comunicação por vector de estado é utilizada para aceder à informação sobre o sistema (Jackson 1983).

O diagrama de especificação do sistema permite representar a cardinalidade existente na comunicação entre as instâncias dos diversos processos. Esta representação obtém-se através de uma barra dupla no lado do processo que possui o relacionamento 'muitos'. Os quatro tipos de relacionamentos representáveis são: um para um, um para muitos, muitos para muitos e muitos para um (Cameron 1986).

Note-se que cada processo utilizado no diagrama de especificação do sistema é descrito através de um diagrama de estrutura do processo[10] e os processos do modelo possuem um conjunto de atributos, que definem o seu estado.

Considera-se que o diagrama de especificação do sistema apresenta os resultados de uma forma concisa, mas torna-se de difícil leitura caso exista muita comunicação entre

[10] *Process Structure Diagram* (PSD).

processos (Hull e O'Donoghue 1993). Para Cameron, Campbell e Ward (1991), esta técnica apresenta uma fraqueza importante, pois não mostra as interacções existentes entre instâncias da mesma entidade.

3.2.2.1.2. Orientadas aos dados

- **Diagrama de Entidades e Relacionamentos**[31]

Um Diagrama de Entidades e Relacionamentos (DER) é um modelo em rede, que descreve a organização dos dados armazenados num sistema, com um elevado nível de abstracção (Yourdon 1989). Um DER, apresenta o modo como os dados estão logicamente agrupados e os relacionamentos entre estes agrupamentos de dados (Skidmore, Farmer e Mills 1992).

As contribuições principais para a introdução dos DER, foram dadas por Chen (1976) e Martin (1982). Infelizmente os DER não estão standardizados, isto é, não existe nenhuma versão da notação, oficial e universalmente reconhecida (Ghezzi, Jazayeri e Mandrioli 1991).

Apesar das diversas notações existentes, grande parte dos autores identifica um conjunto de componentes principais: entidades, relacionamentos e atributos (Chen 1990), (Martin e McClure 1991) e (Pressman 1992).

Seguindo a notação definida por Chen (1990) considera-se que uma entidade é um objecto - pessoa, coisa, local ou evento - sobre o qual deve ser armazenada informação. Uma entidade representa-se por um rectângulo num DER.

Um relacionamento é representado por um losango e linhas conectando-o às entidades que se relacionam. O grau de um relacionamento define o número de entidades associadas no relacionamento. A conectividade de um relacionamento especifica o mapeamento das ocorrências das entidades associadas no relacionamento. Os tipos básicos de conectividade são: um-para-um, um-para-muitos e muitos-para-muitos.

[31] *Entity-Relantionship Diagram.*

A generalização/especialização ocorre quando uma entidade é dividida por diferentes valores de um atributo comum, por exemplo, a entidade empregado é uma generalização das entidades empregado a prazo e empregado efectivo.

A existência de uma entidade pode depender da existência de outra entidade. Neste caso, a entidade dependente denomina-se entidade fraca e representa-se normalmente por um rectângulo de duas linhas.

Verifica-se que as entidades e os relacionamentos têm propriedades que podem ser expressas em termos de pares atributo-valor. A sua representação depende da notação utilizada.

Considera-se que os DER fornecem ao analista uma notação simples e concisa, que facilita a análise e modelação dos dados de um sistema (Pressman 1992). Verifica-se que são muito utilizados na prática, devido:

- ao seu carácter diagramático;
- facilidade de compreensão por não especialistas,
- descrição dos dados de um modo lógico, independente da implementação.

Segundo Sage e Palmer (1990) a maior vantagem dos DER é que apresenta uma perspectiva realista das entidades de dados de que o sistema necessita.

- **Diagrama NIAM**[32]

O diagrama NIAM[33] tem dois objectivos principais, apresentar um quadro simples e claro do universo do discurso para os utilizadores e apresentar uma especificação formal da base de conhecimentos.

O NIAM[34] denomina-se também modelação orientada aos factos, já que o desenvolvimento se inicia com exemplos familiares de factos relevantes, expressos em termos de factos elementares. O método NIAM, deve-se aos trabalhos de Nijssen (1985) e Falkenberg (1982).

[32] *Nijssen Information Analysis Methodology*.
[33] Diagrama do esquema conceptual
[34] Segundo Nijssen e Halpin (1989) este método cobre a etapa de análise e concepção do software.

Pode-se considerar que o esquema conceptual compreende três secções: tipos de factos, restrições e regras de derivação (Nijssen e Halpin 1989).

Na secção tipos de factos indicam-se os tipos de entidades permitidos no universo do discurso, o modo como estes tipos de entidades são referenciados e os seus diferentes papeis.

As restrições podem ser:

- Estáticas, aplicam-se a todos os estados do modelo conceptual;
- Dinâmicas, não permitem determinadas transições entre estados.

A secção regras de derivação, apresenta uma lista de funções, operadores e regras que podem ser utilizadas para derivar informação não explicitamente armazenada. A derivação envolve o cálculo matemático e a inferência lógica.

O método NIAM, aconselha o desenvolvimento através de uma sequência de nove passos bem definidos. Os três primeiros referem-se à identificação dos tipos de factos e os seguintes acrescentam restrições aos tipos de factos armazenados.

A notação do diagrama NIAM, inclui:

- Tipo entidade ⇨ elipses com a identificação do tipo de entidade e o modo de referência;
- Papeis desempenhados ⇨ rectângulos com a descrição dos respectivos papeis.

Segundo Nijssen e Halpin (1989, p.32), "apesar de similar ao modelo de entidades e relacionamentos de Chen (1976), em alguns aspectos, a técnica NIAM, permite uma abordagem mais simples e natural à modelação de dados".

Considera-se que o diagrama NIAM apresenta os seguintes pontos fortes:

- Utilização de apenas uma estrutura de dados (o tipo facto), mas vários tipos de restrições,
- Diagramas com significado visual simples;
- baseia-se inicialmente em exemplos de factos relevantes, expressos em linguagem natural.

3.2.2.1.3. Orientadas ao comportamento

- **Diagrama de Transição de Estado**[35]

Um diagrama de transição de estado (DTE), é uma das notações existentes para descrever uma máquina de estados finitos (McDermid 1991).

Define-se uma máquina de estados finitos como uma máquina hipotética que pode estar apenas em um, de um determinado número de estados, em qualquer instante. Quando é recebida uma determinada entrada apropriada a máquina move-se de um estado para outro e pode gerar saídas. Tanto a saida (S), como o estado seguinte (E_S) são funções que dependem apenas do estado corrente (E_C) e da entrada (Ent). Assim obtém-se (Davis 1990):

$$E_S = F(E_C, Ent)$$
$$S = G(E_C, Ent)$$

Os conceitos principais envolvidos nos DTE, são os eventos, que representam um estimulo externo e os estados que representam valores dos atributos das entidades (Rumbaugh et al 1991) Um DTE é simplesmente um grafo, em que os nodos representam estados e os arcos representam as transições. Normalmente referencia-se nos arcos as entradas e saidas da transição (McDermid 1991).

Segundo Ghezzi, Jazayeri e Mandrioli (1991), os DTEs são uma técnica simples, importante e muito utilizada na descrição do comportamento de um sistema.

- **Diagrama de Ciclo de Vida de Entidade**[36]

O diagrama de ciclo de vida de entidade[37] (DCVE), preocupa-se com a identificação dos eventos que afectam uma entidade e com a sequência temporal destes eventos (Skidmore, Farmer e Mills 1992). O DCVE, permite realizar a análise das alterações sofridas por uma entidade ao longo do tempo, isto é, o seu comportamento.

[35] *State Transition Diagram.*
[36] *Entity Life History.*
[37] No JSD (Jackson 1983) denomina-se *Process Structure Diagram.*

Os DCVEs são diagramas de estrutura em árvore, lidos da esquerda para a direita. O nodo raiz representa um tipo entidade e contém o nome da entidade. Apenas as acções podem ser folhas e o resto da árvore, utiliza-se para agrupar as acções e obedecer ás convenções do diagrama (Sutcliffe 1988).

As estruturas fundamentais de um DCVE são: sequência, selecção e iteração. As sequências e selecções podem ter dois ou mais componentes, as iterações apenas possuem um componente. As selecções e iterações são representadas, respectivamente pelos simbolos "0" e "*", colocados no canto superior direito dos componentes constituintes (Cameron 1986) Esta notação é normalmente designada notação-Jackson.

O DCVE é uma forma gráfica de representar expressões regulares (Cameron, Campbell e Ward 1991).

Esta técnica deve-se a Jackson (1983), embora seja utilizada em outros métodos, por exemplo, o SSADM (Downs, Clare e Coe 1992).

3.2.2.1.4. Orientadas aos objectos

O desenvolvimento de software orientado aos objectos baseia-se em três principios fundamentais: encapsulação, herança e classificação (Sutcliffe 1991), (Loy 1990) Os objectos são uma abstracção de partes do mundo real e modelam unidades compostas de estrutura e actividade. Segundo Sernadas (1993), um objecto define-se como uma unidade de *encapsulação*, com um interface de serviços (o seu conteúdo interno encontra-se escondido) Os objectos devem possuir propriedades genéricas, suportando a reutilização através da *herança* de propriedades das superclasses para as subclasses. O conceito da *classificação* está associado à identificação dos objectos e classes e dos seus relacionamentos (Booch 1991).

As técnicas de representação orientadas aos objectos são de dificil classificação, quanto ao objectivo de cobrir a etapa de análise dos requisitos do software, concepção do software ou ambas. É particularmente complexo determinar onde termina a análise e onde começa a concepção do software (Monarchi e Puhr 1992).

Considera-se que o objectivo da análise orientada aos objectos é modelar a semântica do problema, em termos de objectos distintos mas relacionados. A concepção orientada aos objectos modela os detalhes do comportamento de cada objecto e reexamina as classes do domínio do problema para melhorar a reutilização e tirar partido da herança (Fichman e Kemerer 1992).

Existe um elevado número de métodos de desenvolvimento orientados aos objectos, embora seja uma abordagem relativamente recente. Segundo Monarchi e Puhr (1992), estes métodos podem classificar-se pelo modo como incorporam outros paradigmas, obtendo-se.

- *abordagem combinativa*, que utiliza técnicas orientadas aos objectos, funcionais e/ou orientadas ao comportamento para modelar o sistema. Os métodos de Rumbaugh et al. (1991) e Shaeler e Mellor (1988), situam-se nesta categoria;

- *abordagem adaptativa*, utiliza técnicas existentes de uma forma inovadora, incorporando a orientação aos objectos. Por exemplo, o método de Bailin (1989) e o método de Henderson-Sellers e Constantine (1991),

- *abordagem "pura"*, utiliza novas técnicas orientadas aos objectos para modelar a estrutura do objecto, a sua funcionalidade e o comportamento. Os métodos de Coad e Yourdon (1991) e Booch (1991), encontram-se nesta categoria.

Apresenta-se em seguida uma breve descrição das técnicas orientadas aos objectos principais, incorporadas nos métodos Rumbaugh et al. (1991), Bailin (1989) e Coad e Yourdon (1991), representativos de cada uma das categorias referidas.

Pretende-se também classificar as técnicas de representação orientadas aos objectos, quanto à(s) perspectiva(s) que pretendem modelar, ou seja,

- serviços (orientação aos processos);
- objectos, atributos e relacionamentos (orientação aos dados);
- comunicação e temporização/controlo(orientação ao comportamento).

- **Diagrama de Fluxos de Dados-Entidades**[38]

O Diagrama de Fluxos de Dados-Entidades (DFDE) é a técnica de representação central do método Especificação Orientada aos Objectos[39] (EOO) desenvolvido por Bailin (1989). Este método é a resposta à incompatibilidade perceptível entre a análise estruturada convencional e a concepção orientada aos objectos.

O método de Bailin, assemelha-se à análise estruturada moderna (Yourdon 1989) já que a decomposição do sistema é realizada utilizando uma notação semelhante à dos DFDs. O produto final é um conjunto de DERs e uma hierarquia de DFDEs (Fichman e Kemerer 1992).

Bailin (1989, p.608) afirma que o suporte da análise estruturada é o seguinte princípio de agregação "as funções são agrupadas se são passos constituintes da execução de uma função de nível mais alto", enquanto numa perspectiva orientada aos objectos, considera-se que o princípio de agregação tem a forma: "as funções são agrupadas se operam sobre a mesma abstracção de dados".

A técnica DFDE, não se baseia na análise das transformações de entradas em saídas (processos), mas antes no conteúdo dos objectos[40]. Este conteúdo apresenta três perspectivas: as estruturas de dados que definem os objectos; o estado subjacente de um processo à medida que evolui no tempo e a parte de um processo que é persistente durante vários ciclos de execução.

Os DFDEs são semelhantes aos DFDs convencionais, excepto os nodos que são de duas categorias: objectos e funções. Todas as funções ocorrem no contexto de um objecto, isto é, são executadas ou sofridas por um objecto.

Segundo Bailin (1989), distinguem-se duas classes de objectos: activos e passivos. Considera-se que os objectos activos realizam operações (sobre eles mesmos ou sobre outros objectos) suficientemente importantes, para serem analisados em detalhe durante a fase de análise, enquanto os objectos passivos têm menor importância e podem ser tratados como

[38] *Entity-Data Flow Diagram* (EDFD)
[39] *Object-Oriented Specification* (OOS).
[40] Bailin (1989) utiliza as palavras entidade ou objecto sem distinção.

caixas pretas até à fase de concepção do software. Os objectos activos são representados por nodos nos DFDEs, enquanto os objectos passivos são fluxos ou depósitos de dados.

Os DFDEs decompõem cada objecto activo ou em sub-objectos ou em funções executadas pelo objecto (ou numa combinação dos dois). As funções podem ser decompostas em sub-funções. O nivel zero contém apenas objectos, assegurando-se assim que todas as funções pertencem a um objecto. A distinção, em termos de notação, entre objectos e funções é feita colocando-se parênteses nos objectos.

O DFDE, permite modelar os serviços, identificar os objectos e estabelecer as comunicações entre os objectos. No método EOO, a definição dos relacionamentos e dos atributos é tratada com os DERs.

- **Diagrama de Classes e Objectos[41] (D. Coad)**

O diagrama de classes e objectos possui cinco camadas que são desenvolvidas incrementalmente, durante cada uma das etapas previstas no método análise orientada aos objectos[42] proposto por Coad e Yourdon (1991).

Segundo Coad e Yourdon (1991, p.57), "um objecto é uma encapsulação e uma abstracção: uma encapsulação de atributos e serviços exclusivos sobre esses atributos; uma abstracção do espaço do problema, representando uma ou mais ocorrências de *algo* no espaço do problema". Partindo desta definição o diagrama de classes e objectos constroi-se, através da seguinte sequências de etapas:

1. *Identificação de objectos*. Consiste na análise do sistema, para localizar estruturas, outros sistemas, dispositivos, eventos, papeis, e locais;

2. *Definição de estruturas*. Procuram-se relacionamentos entre as classes e representam-se ou como estruturas de generalização-especialização ou estruturas do tipo tôdo-partes;

[41] *Class and object diagram*.
[42] *Object-Oriented Analysis* (OOA).

3. *Identificação de áreas dependentes*[43]. Examina-se os objectos de alto-nível, dentro das hierarquias tôdo-partes e marcam-se como candidatos a áreas dependentes. Refina-se as áreas dependentes para minimizar as interdependências entre elas.

4. *Definição de atributos*. Identificam-se as características atómicas dos objectos, como seus atributos. Procura-se também relacionamentos associativos entre objectos e determina-se a cardinalidade desses relacionamentos. Esta cardinalidade é modelada com a notação familiar de "pés de galinha"

5. *Definição de serviços*. Para cada objecto identifica-se todos os serviços que executa. Mostra-se também as conexões de mensagem existentes, através de arcos ligando os objectos.

O diagrama de classes e objectos, permite identificar os objectos, os seus atributos, relacionamentos e comunicações. Verifica-se também que a etapa cinco permite identificar os serviços de cada objecto. A definição completa dos serviços é obtida através da utilização dos *diagramas de serviços*.

Considera-se que o diagrama de classes e objectos é simples, eficaz e suficiente para modelar a maior parte dos sistemas no seu nível estático, contudo não é suficiente para modelar os aspectos dinâmicos (temporização e controlo). Segundo Coad e Yourdon (1991), os aspectos dinâmicos de um sistema devem ser definidos através dos *diagramas de estado do objecto*, uma variação dos diagramas de transição de estados.

- **Diagrama de Objectos[44] (D. Rumbaugh).**

O diagrama de objectos é uma das técnicas utilizadas pelo método de desenvolvimento de software, denominado Técnica de Modelação de Objectos[45] e apresentado por Rumbaugh et al. (1991).

[43] *Subject Areas*.
[44] *Object Diagram*.
[45] *Object Modeling Technique* (OMT).

A técnica de modelação de objectos, prevê a utilização de três modelos - objectos, dinâmico e funcional - para representar as diferentes perspectivas de um sistema real e especificar os requisitos do software.

Segundo Rumbaugh et al. (1991), o desenvolvimento do diagrama de objectos é feito através de uma sequência de passos:

- Identificação de objectos e classes;
- Identificação de associações (incluindo agregações) entre objectos;
- Identificação de atributos dos objectos e ligações;
- Organizar e simplificar as classes utilizando a herança;
- Agrupar classes em módulos.

Considera-se que um módulo é um conjunto de classes que agrupam um sub-conjunto lógico do modelo global. É um mecanismo que permite gerir a complexidade apresentada pelos grandes sistemas (Monarchi e Puhr 1992).

O diagrama de objectos permite modelar apenas a perspectiva dos dados enquanto a perspectiva funcional e o comportamento são modelados com outras técnicas, DFDs e DTEs respectivamente (Rumbaugh et al. 1991). Verifica-se, que o diagrama de objectos permite definir os objectos, os relacionamentos e atributos, e também identificar os serviços de cada objecto.

3.2.2.2. Técnicas de representação textuais

As técnicas textuais, baseiam-se numa linguagem textual, com carácter informal ou semi-formal, para descrever as propriedades do sistema.

O exemplo mais comum de uma técnica de representação textual é a linguagem natural. A utilização do português, permite especificar todas as propriedades requeridas, não sendo necessário nenhum treino especial para utilizar este tipo de especificações. Considera-se contudo que as especificações resultantes, são quase sempre ambíguas, incompletas, não apresentam uma estrutura bem definida e são incompreensíveis, devido ao seu volume. Outra forte desvantagem, refere-se ao aspecto de não ser possível nenhum processamento automático, para além das verificações sintáticas no seu nível mais superficial (Zave 1990).

A linguagem natural é uma técnica informal, contudo existem diversas técnicas textuais semi-formais, por exemplo: Linguagem de declaração dos requisitos[46] de Bell e Bixter (1976); Linguagem de especificação orientada aos processos, aplicativa e interpretável[47], desenvolvida por Zave (1982); Linguagem de concepção de programas[48].

Apresenta-se na próxima secção, as principais características da linguagem de concepção de programas.

- **Linguagem de Concepção de Programas.**

A linguagem de concepção de programas (LCP) é actualmente o standard *de facto* para a especificação dos detalhes de concepção dos módulos de software (McDermid 1991) e (Davis 1988). Contudo segundo Davis (1990) esta linguagem pode ser utilizada para especificar os requisitos do software.

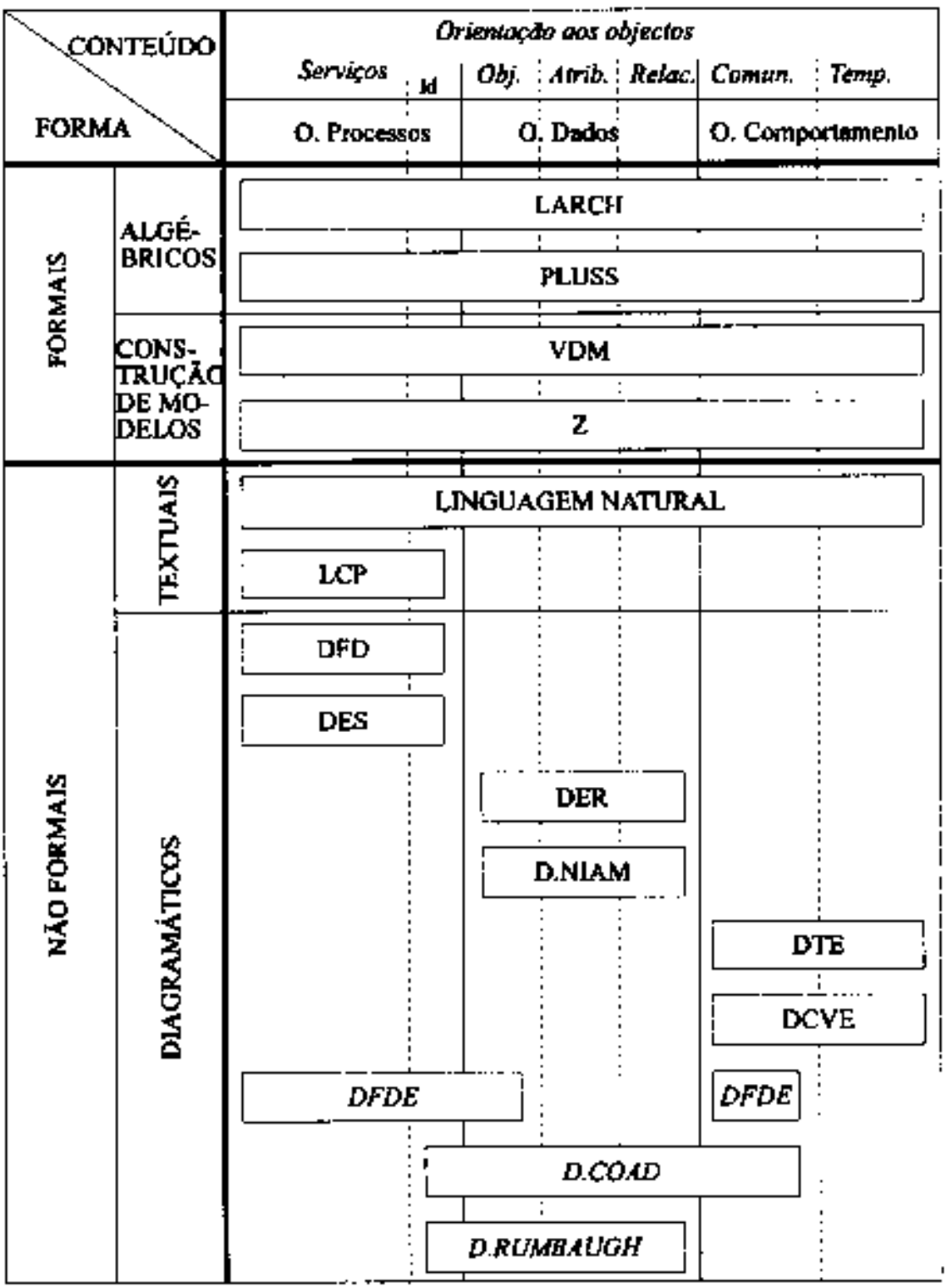
A linguagem de concepção de programas, designa-se também por português-estruturado ou pseudo-código e é simplesmente linguagem natural estruturada, com um número restrito de palavras-chave reservadas, com significado especial, por exemplo, *se*, *senão*, *repetir*, etc. Existem diversas variações desta linguagem, contudo verifica-se que todas contêm palavras-chave para representar as três estruturas básicas principais: sequência, repetição e condição. A versão mais utilizada desta linguagem foi proposta por Caine e Gordon (1975)

Segundo McDermid (1991), a linguagem de concepção de programas, sofre de muitos dos problemas ligados à linguagem natural, nomeadamente: ambiguidade, inconsistências e especificações incompletas ou demasiado detalhadas.

Esta linguagem permite especificar os requisitos funcionais de um sistema, embora seja necessária extrema prudência, para não entrar em detalhes da concepção (Davis 1990)

[46] *Requirement Statement Language* (RSL).
[47] *Process-Oriented, Applicative, and Interpretable Specification Language* (PAISLey).
[48] *Program Design Language (PDL).*

3.2.3. Classificação (quanto à forma e conteúdo) das técnicas de especificação dos requisitos do software



3.3. Concepção do software. Classificação das técnicas de representação.

Definiu-se a concepção do software, como o processo de desenvolvimento de uma solução que satisfaça os requisitos do software. O produto final deste processo, denomina-se especificação do software[49]. Nesta secção pretende-se classificar as técnicas de especificação do software, quanto à sua forma e conteúdo

No que se refere à forma, utiliza-se uma classificação semelhante à definida na secção anterior Dividem-se as técnicas de especificação do software como **não formais** ou **formais**[50].

Quanto ao conteúdo existem algumas opiniões divergentes. Apresenta-se em seguida a perspectiva de alguns autores.

Hodgson e Dunne (1990), consideram que o desenvolvimento do domínio-solução, tal como a definição do problema, deve cobrir as três perspectivas características - dados, funções e comportamento - de um sistema.

Segundo Sage e Palmer (1990) as técnicas de especificação do software podem ser divididas em:

- Técnicas de especificação da arquitectura do software,
    - Técnicas convencionais (decomposição);
        - Abordagem orientada aos fluxos de dados;
        - Abordagem orientada às estruturas de dados;
    - Abordagem orientada aos objectos (composição),
- Especificação detalhada do software.

Para Pfleeger (1991), utilizam-se duas estratégias principais para a concepção do software:

- Decomposição (concepção baseada em funções);
- Composição:
    i. baseada em objectos;
    ii. baseada nos dados.

[49] Ver secção 2.3.2.
[50] Ver definições na secção 3.2.

Finalmente, Pressman (1992) classifica as técnicas de especificação do software, como endereçando:

- concepção dos dados;
- concepção da arquitectura dos módulos;
- concepção dos objectos;
- concepção dos procedimentos.

Considera-se que as técnicas de especificação do software, quanto ao conteúdo, podem ser classificadas em: **orientadas aos objectos (composição)** e **abordagem convencional (decomposição).**

Definiu-se a análise orientada aos objectos, como descrevendo o dominio do problema, isto é, representando objectos semânticos[51]. Segundo Monarchi e Puhr (1992) a concepção **orientada aos objectos**, consiste em refinar os objectos semânticos e incluir outros tipos de objectos dependentes do domínio-solução: objectos do interface, objectos da aplicação e objectos de base e utilitários.

Considera-se que os objectos do interface definem o interface com os utilizadores, os objectos da aplicação representam os dispositivos e mecanismos de controlo especificos para o sistema e os objectos de base e utilitários serão os objectos componentes, independentes da aplicação mas utilizados na sua construção.

Conclui-se então, que as técnicas de especificação do software orientadas aos objectos, tal como as técnicas de representação dos requisitos devem cobrir as três perspectivas básicas de um sistema (funções, dados e comportamento). Em termos do desenvolvimento orientado aos objectos, significa.

- **serviços**;
- **objectos**, **atributos** e **relacionamentos**;
- **comunicação, temporização / controlo**.

A **abordagem convencional** à especificação do software é uma estratégia intimamente ligada às linguagens de terceira geração e divide-se em: concepção das **estruturas de dados**, concepção da **arquitectura dos módulos** e concepção dos **detalhes dos procedimentos**

[51] Objectos com significado no dominio do problema.

A concepção das **estruturas de dados**, consiste na definição do esquema conceptual e do esquema interno dos dados. Segundo Nijssen e Halpin (1989) o esquema conceptual descreve a estrutura do universo do discurso e o esquema interno, especifica o armazenamento físico e estruturas de acesso utilizadas para implementar o modelo conceptual de dados, de um modo eficiente. A concepção das estruturas de dados, pode-se dividir em dois níveis, o nível conceptual (lógico) e o nível interno (físico).

A concepção da **arquitectura dos módulos**, apresenta de uma forma modular a estrutura hierárquica do programa, representando os relacionamentos de controlo entre módulos. Considera-se que faz a ligação entre a estrutura do programa e as estruturas de dados, permitindo representar o fluxo de dados no programa (Pressman 1992).

A concepção dos **procedimentos**, deve especificar os detalhes dos módulos de uma forma não ambígua.

Em resumo, obtém-se:

Técnicas de especificação do software

FORMA

- Formais
- Não formais

CONTEÚDO

- Objectos
  - Serviços
  - Objectos, atributos e relacionamentos
  - Comunicação e temporização / controlo
- Abordagem convencional
  - Arquitectura dos módulos
  - Estrutura dos dados
  - Detalhes dos procedimentos

3.3.1. Técnicas formais para a especificação do software

Segundo Wing (1990, p.13), "duas das actividades mais importantes durante a concepção do software são a decomposição e o refinamento". Nesta perspectiva os métodos formais adaptam-se perfeitamente à tarefa de concepção de software.

Considera-se que o refinamento é a produção de uma implementação a partir da especificação abstracta (McDermid 1991), (Oliveira 1992) através da aplicação de um número de passos de desenvolvimento, cada um centrado numa decisão de concepção bem fundamentada. Jones e Shaw (1990) consideram que o processo de refinamento envolve a introdução controlada de detalhes relacionados com a partição do espaço do problema, selecção de algoritmos abstractos, representação de dados, decomposição algorítmica e implementação.

Grande parte do refinamento consiste no processo denominado 'reificação', que significa, tornar mais concreto, e, muitas vezes, o termo reificação é utilizado em vez de refinamento (McDermid 1991).

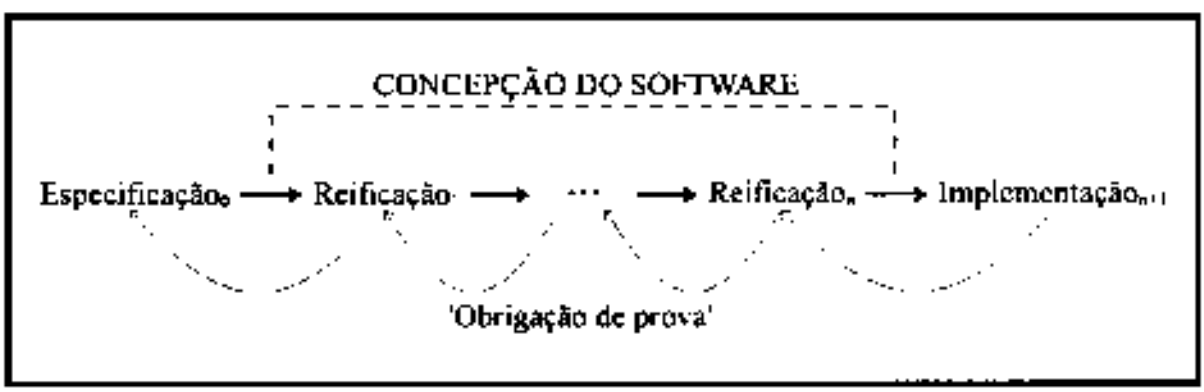


Figura 21 - Processo de refinamento (adaptado de Jones e Shaw [1990]).

A figura 21, apresenta o processo de refinamento de uma forma simplificada. Pode-se no entanto verificar que o conceito de reificação, fornece uma estrutura formal para suporte da etapa de concepção do software (Sinderen, Pires e Vissers 1992)

Refira-se que a implementação tem que incluir uma descrição detalhada (concreta) dos dados utilizados pelo software e uma descrição detalhada (concreta) dos algoritmos utilizados para executar as operações (McDermid 1991). O desenvolvimento formal de software é feito

através do refinamento de dados, do refinamento de operações (Oliveira 1992) ou da decomposição.

Define-se decomposição como o processo de dividir o sistema em módulos menores, que podem ser reificados independentemente. A especificação formal permite capturar com precisão as ligações entre módulos, adaptando-se perfeitamente a esta estratégia de concepção (Wing 1990).

Para assegurar que uma série de passos de refinamento/decomposição preservam a correção, isto é, satisfazem a especificação do nível mais alto, existe a obrigação de provar que cada transformação modela correctamente a especificação anterior. A esta verificação denomina-se "obrigação de prova"[52] (Jones e Shaw 1990).

Pode-se concluir, que o desenvolvimento formal, através do refinamento e decomposição permite conceber a arquitectura dos módulos, as estruturas de dados e os detalhes dos procedimentos.

3.3.2. Técnicas de especificação do software não formais

3.3.2.1. Abordagem convencional

3.3.2.1.1. Concepção da arquitectura dos módulos

- **Diagrama de Estrutura**

O diagrama de estrutura (DE) é a técnica base do desenho estruturado[53], apresentado por Stevens, Myers e Constantine (1974) e Yourdon e Constantine (1978).

O diagrama de estrutura é uma árvore ou estrutura hierárquica que define a arquitectura global de um programa, mostrando os módulos e as suas interrelações (Martin e McClure 1991). Os seus componentes básicos são: módulos, ligações entre módulos e comunicação entre módulos.

[52] Ver figura 21.
[53] *Structured Design*.

Para Page-Jones (1988, p.45) um módulo "é uma colecção de instruções de um programa com quatro atributos básicos: entradas e saidas, função, lógica e dados internos". Um diagrama de estrutura permite representar a perspectiva externa dos módulos, isto é, a sua função e as entradas e saidas

Num diagrama de estrutura, o módulo representa-se por um rectângulo dentro do qual está contido o seu nome. Um módulo pré-definido é mostrado graficamente através da adição de linhas internas paralelas às linhas verticais.

Os módulos são interrelacionados por uma estrutura de controlo O diagrama de estrutura apresenta os interrelacionamentos, através da organização dos módulos em níveis e da ligação dos niveis com setas. Uma seta unindo dois módulos significa que em determinado ponto da execução o controlo do programa é passado de um módulo para o outro no sentido da seta (Martin e McClure 1991).

A comunicação entre os módulos representa-se com o símbolo ' ○⟶ ', para os dados e com o simbolo ' ●⟶ ' para a informação de controlo. A direcção da seta indica qual o módulo que envia a informação

Finalmente, pode-se descrever a lógica dos procedimentos a partir dos seguintes construtores básicos: selecção (coloca-se um losango sobre as ligações de invocação aos módulos pertencentes à selecção) e iteração (utiliza-se o simbolo ' ↻ ' na invocação ao módulo chamado várias vezes).

- **Diagrama de Estrutura do Processo**

O diagrama de estrutura do processo (DEP) baseia-se na notação de Jackson, já apresentada[54] e a técnica de desenvolvimento é derivada dos procedimentos utilizados na programação estruturada de Jackson[55] (Ingervaldsson 1979).

Os métodos JSD (Jackson 1983) e SSADM (Downs, Clare e Coe 1992), realizam a especificação do software através desta técnica de representação Verifica-se que no SSADM,

[54] Secção 3.2.2.1.3.
[55] *Jackson Structured Programming* (JSP).

o diagrama de estrutura do processo é utilizado na etapa de desenho lógico dos processo[56], resultando na produção dos modelos do processo para actualizar e inquirir as estruturas de dados[57].

A tarefa central na preparação de um DEP é a combinação das estruturas de dados de entrada e saída. Basicamente, a estrutura necessária do processo tem que incorporar tanto as estruturas de dados de entrada como de saída, verificando-se que a estrutura do processo é de facto determinada pelas estruturas de dados (Skidmore, Farmer e Mills 1992).

Segundo o SSADM, o desenvolvimento dos diagramas de estrutura do processo é feito através da seguinte sequência de tarefas:

- Preparar as estruturas de dados de entrada e saída (notação de Jackson);
- Combinar as estruturas de dados para obter o DEP inicial;
- Derivar as operações apropriadas dos DCVEs e alocá-las ao DEP;
- Identificar e alocar condições ao DEP

As correspondências entre as estruturas de dados podem ser encontradas através de um conjunto de regras simples, desenvolvidas por Burgess (1987). Quando se combinam as estruturas de dados podem ocorrer 'colisões de estrutura', que se dividem em três tipos

1. *colisão de ordem*, surge quando a sequência dos componentes é diferente nas duas estruturas,

2. *colisão de fronteira*, ocorre quando o agrupamento de elementos de dados de entrada é diferente do requerido pelo mecanismo de saída;

3. *colisão entre folhas*, ocorre quando os eventos para uma entidade estão separados pelos eventos para outra entidade.

Considera-se que o DEP ajuda na formação de ideias precisas sobre a concepção do software, revelando-se contudo fraco no caso da lógica complexa de programas (Martin e McClure 1991).

[56] *Logical Process Design*.
[57] *Enquiry Process Model* (EPM) e *Update Process Model* (UPM).

Segundo Hull e O'Donoghue (1993) o DEP orienta a definição e organização dos módulos, mas não os seus procedimentos internos. Considera-se então que permite apresentar a arquitectura dos módulos

3.3.2.1.2. Concepção das estruturas de dados

Considerou-se que as técnicas de especificação das estruturas de dados, devem ser divididas em dois níveis, o nível conceptual (lógico) e o nível interno (físico). Nesta secção analisa-se o diagrama de entidades e relacionamentos e o diagrama NIAM, no nível conceptual e o modelo relacional no nível interno.

Independentemente, da técnica de especificação do software analisada, considera-se sempre que o objectivo da etapa de concepção do software é o desenvolvimento de uma *solução* que satisfaça os requisitos definidos

O diagrama de entidades e relacionamentos[58] permite modelar o problema na perspectiva dos dados. Contudo, após a definição dos requisitos de dados utilizando os DERs, são aplicadas diversas transformações para definir uma possível solução. Estas transformações podem ser do tipo 'receita', isto é aplicação de um conjunto bem definido de passos, o exemplo mais comum é a normalização ou do tipo 'informal', ou seja, modificações no DER devido a um conjunto variável de factores - experiência do analista, necessidade de protecção, velocidade, etc - que se traduz por exemplo na introdução de redundância.

Verifica-se a apetência dos DERs para a análise dos requisitos e concepção do software, embora seja complexo definir com precisão onde termina uma etapa e começa a outra.

O diagrama NIAM[59], é desenvolvido através de uma sequência de nove passos bem definidos e segundo Nijssen e Halpin (1989), este método permite cobrir os aspectos relacionados com a análise dos requisitos e concepção do software.

Considera-se que as três primeiras etapas, cujo objectivo é identificar os tipos de factos permitem especificar os requisitos do software. A concepção do software é realizada

[58] Ver secção 3.2.2.1.2.
[59] Descrito na secção 3.2.2.1.2.

através das sete etapas seguintes e de um conjunto de transformações que permitem obter esquemas conceptuais equivalentes e logicamente flexibilizar o desenho. Na prática estas transformações finais permitem reformatar o esquema conceptual na perspectiva de obter uma implementação mais eficiente (Nijssen e Halpin 1989).

- **Modelo relacional**

O modelo relacional (MR) foi inicialmente apresentado por Codd (1970) verificando-se que os fundamentos e grande parte dos conceitos apresentados por este autor ainda se mantêm válidos actualmente.

Um modelo relacional é constituido por uma colecção de relações normalizadas, isto é, a única estrutura de dados permitida é a tabela ou relação, que é formada por um conjunto de tuplos (Nijssen e Halpin 1989).

Considera-se que uma relação é uma tabela bi-dimensional, composta por colunas e linhas de dados (tuplos). Cada coluna apresenta um atributo da relação e um tuplo é uma lista de valores dos atributos. O domínio representa o conjunto de valores que determinado atributo pode tomar.

Define-se chave candidata como um atributo ou conjunto de atributos que não podem ter valores duplicados em nenhuma ocorrência da relação, ou seja, o seu valor é sempre suficiente para identificar um tuplo. A chave candidata seleccionada para identificar univocamente a relação designa-se chave primária. Finalmente uma chave externa é uma chave candidata incluida noutra relação e que representa o relacionamento entre as duas relações

Esta notação permite especificar o esquema interno dos dados, sendo óbvia a sua aproximação a uma implementação especifica (bases de dados relacionais)

3.3.2.1.3. Concepção dos detalhes dos procedimentos

A base de todas as técnicas de concepção dos detalhes do software, encontra-se nos trabalhos de Dijkstra (1976). Este autor propõe a utilização de um conjunto de construtores lógicos, a partir dos quais qualquer programa pode ser especificado. Os construtores são a sequência, a repetição e a condição.

Existem diversas técnicas para especificar os detalhes dos procedimentos, por exemplo, diagramas de Nassi-Shneiderman, fluxogramas, árvores e tabelas de decisão, linguagem de concepção de programas e diagramas de acção. Nesta secção analisam-se os diagramas de acção e a linguagem de concepção de programas.

A linguagem de concepção de programas[60] quando utilizada na descrição dos detalhes dos procedimentos deve apresentar as seguintes caracteristicas (Pressman 1992):

- um conjunto fixo de palavras-chave que forneça os construtores básicos, declarações de dados e caracteristicas de modularidade;
- linguagem natural que permita descrever caracteristicas de processamento;
- facilidades de declaração de dados, incluindo estruturas de dados simples (arrays, strings, ...) e complexas (listas ligadas, árvores, .),
- definição de subprogramas e técnicas de invocação que suportem vários modos de descrição do interface.

Actualmente verifica-se que na concepção dos detalhes dos procedimentos é sempre utilizada uma linguagem de programação de alto nível como base para a linguagem de concepção de programas.

- **Diagrama de Acção**

Segundo Martin e McClure (1991), o diagrama de acção permite especificar a arquitectura dos módulos e os detalhes dos procedimentos.

Os parênteses rectos são os blocos básicos de construção dos diagramas de acção, representando os módulos. Dentro dos parênteses apresenta-se a sequência de operações. Pode-se representar um (ou vários) parêntese dentro de outro permitindo assim definir a estrutura hierárquica de um programa.

[60] Descrita nas técnicas de representação dos requisitos do software, secção 3.2.2.2.

A notação proposta pelos diagramas de acção, possui diversos construtores. Os principais são:

- *repetição*, através de um parêntese recto com uma linha dupla na parte superior;
- *condição*, utilizando um parêntese com várias divisões;
- *fim condicional*, representa-se por uma seta da direita para a esquerda, atravessando o(s) parêntese(s) pretendido(s);
- *acesso a estruturas de dados*, um rectângulo contendo o nome de um tipo registo precedido por uma acção de acesso aos dados: ler, actualizar, criar e excluir.

Refira-se também que quando se prevê uma determinada linguagem de programação de alto nível para a implementação, pode-se utilizar as primitivas especificas da linguagem de programação no próprio diagrama de acção.

Para Martin e McClure (1991), os diagramas de acção apresentam algumas vantagens sobre outras técnicas de especificação dos detalhes do software, por exemplo, o seu fácil manuseamento, relacionamento simplificado com o modelo de dados, facilidade de implementação em linguagens de programação de alto-nível.

3.3.2.2. Concepção orientada aos objectos

- **Diagrama de Classes e Diagrama de Objectos (Booch)**

O diagrama de classes e o diagrama de objectos constituem as técnicas de representação principais, propostas pelo método Concepção Orientada aos Objectos[61] (Booch 1991). Jacobson et al. (1992, p.487) afirmam que "este método não é realmente um processo, mas antes uma colecção de técnicas e heurísticas que podem ser utilizadas no desenvolvimento orientado aos objectos".

[61] *Object-Oriented Design* (OOD).

Apesar de não apresentar uma sequência precisa de etapas, Booch (1991), enfatiza o carácter iterativo do processo e a criatividade como componentes essenciais da concepção orientada aos objectos. Verifica-se também que Booch (1991) delinea quatro grandes fases que devem ser cobertas durante a construção[62] orientada aos objectos:

1. *Identificação de classes e objectos num determinado nível de abstracção*, envolve a descoberta das abstracções principais no espaço do problema e dos mecanismos que descrevem o comportamento dinâmico dos objectos;

2. *Identificação da semântica das classes e objectos*, envolve o estabelecimento do significado das classes e objectos identificados anteriormente;

3. *Identificação dos relacionamentos;*

4. *Implementação de classes e objectos.*

Segundo Fichman e Kemerer (1992) o diagrama de classes é utilizado para representar as classes e os seus relacionamentos, indicando a arquitectura de classes para o sistema. Os três elementos mais importantes deste diagrama são, as classes, os relacionamentos e os utilinários de classes (dependem do suporte dado pela linguagem de programação).

As classes representam-se com uma forma de 'nuvem', limitada com uma linha a tracejado e com a identificação no seu interior. O relacionamento *herança*, representa-se por uma seta ligando as classes intervenientes e os conectores de duas linhas paralelas indicam que uma classe *utiliza* a informação contida noutra classe. Ao relacionamento *utiliza* (duas linhas paralelas) acrescenta-se um simbolo indicativo da cardinalidade. O diagrama de classes permite representar também os relacionamentos, *instância*, *metaclasse* e *indefinido*.

A documentação de cada classe[63], permite definir os atributos e operações associadas a cada classe, além de outra informação facultativa.

O diagrama de objectos, define uma instância arbitrária de cada classe, representando as operações que serão aplicadas e as mensagens que serão passadas. Um objecto tem a forma de 'nuvem' limitada por um traço continuo, com a sua designação no interior. A notação representa uma comunicação por mensagem através de uma linha unindo os objectos

[62] Considerando-se que a construção inclui a concepção e a implementação (Jacobson et al. 1992)
[63] *Class Diagram Template*.

intervenientes e esta linha de conexão de mensagem entre objectos termina com um quadrado e uma letra que indica o tipo de dados contido numa mensagem. Uma seta paralela à linha de mensagem indica o tipo de sincronização entre os objectos

Associado ao diagrama de objectos, desenvolve-se uma especificação para cada objecto e mensagem[64], permitindo identificar as operações associadas às mensagens e apresentar informações diversas (por exemplo, persistência do objecto, frequência da mensagem, etc).

Verifica-se que o diagrama de classes permite identificar objectos, atributos e relacionamentos e definir as operações. O diagrama de objectos, mostra parte da dinâmica dos objectos permitindo definir a sua comunicação por mensagem.

Refira-se ainda que o método de Booch prevê a utilização de outras técnicas de representação, nomeadamente, diagrama de transição de estados, diagrama de temporização, diagrama de módulos e diagrama de processos.

- **Diagrama de Estrutura Orientado aos Objectos[65]**

O diagrama de estrutura orientado aos objectos (DEOO) é a técnica de representação principal, proposta pelo método Concepção Estruturada Orientada aos Objectos[66]. Este método foi apresentado por Wasserman, Pircher e Muller (1990) e fornece uma notação standard para a concepção do software que suporta a abordagem convencional e a abordagem orientada aos objectos.

Segundo Fichman e Kemerer (1992), o método de concepção estruturada orientada aos objectos, oferece uma notação híbrida que incorpora os conceitos de diversas áreas, nomeadamente:

- diagramas de estrutura da concepção estruturada;
- a notação de Booch (1986) para módulos e tarefas;
- hierarquia de classes e princípios de herança da programação orientada aos objectos;
- conceito de monitores da programação concorrente

[64] *Object and Message Templates.*
[65] *Object-Oriented Structure Chart.*
[66] *Object-Oriented Structured Design* (OOSD).

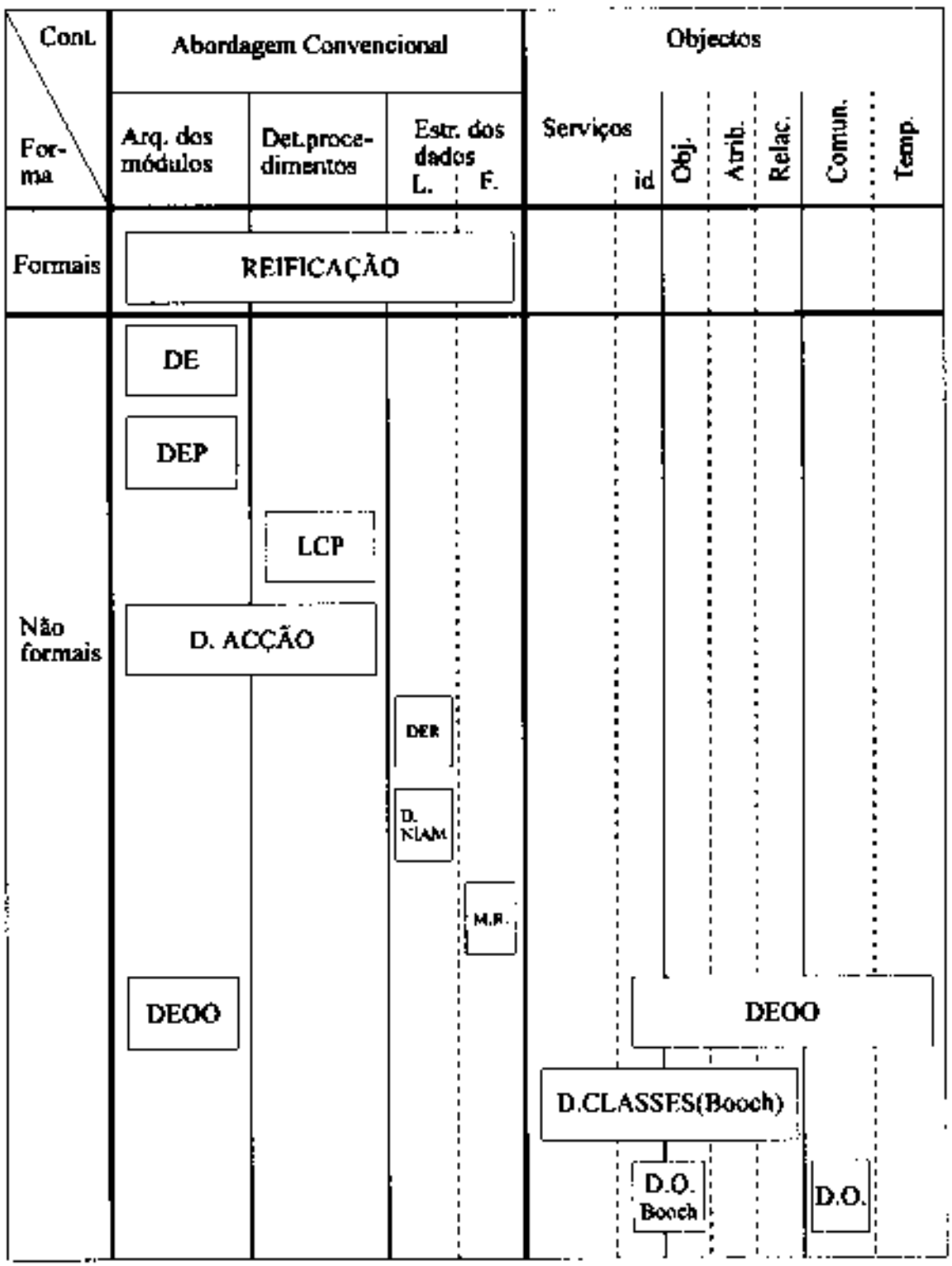
O DEOO, inclui os símbolos básicos dos diagramas de estrutura convencionais[67], ou seja, módulos, estruturas de invocação e passagem de parâmetros, acrescentando-lhe as construções específicas da orientação aos objectos: classes, objectos, métodos, herança, instanciação e definições genéricas. Verifica-se ainda, que a notação permite representar a comunicação por mensagem, a herança múltipla e a concorrência.

Considera-se que a notação permite a utilização exclusiva dos conceitos orientados aos objectos, ignorando a simbologia relacionada com os diagramas de estrutura (Wasserman, Pircher e Muller 1990).

Como conclusão, pode afirmar-se que o DEOO, permite identificar objectos, atributos, serviços e relacionamentos (exclui-se a representação da agregação), bem como a comunicação por mensagens e a temporização/controlo (Monarchi e Puhr 1992). No que se refere à abordagem convencional (decomposição), esta técnica permite representar a arquitectura dos módulos.

[67] Ver secção 3.3.2.1.1.

3.3.3. Classificação (quanto à forma e conteúdo) das técnicas de especificação do software



3.4. Implementação do software. Classificação das linguagens de programação

Considera-se que a implementação do software é a fase do processo de desenvolvimento de software durante a qual a especificação do software é convertida numa linguagem que é executável pelo computador (Thayer e Dorfman 1990) O resultado da implementação do software é a aplicação pronta a ser testada e instalada. Note-se que, nesta fase do processo de desenvolvimento de software, as técnicas de representação denominam-se linguagens de programação. Pretende-se então, definir uma classificação para as linguagens de programação.

Segundo Friedman (1991), as linguagens de programação podem classificar-se de diversos modos:

- pelo nível de complexidade;
- pela área de aplicação para que foi concebida;
- pela geração a que pertence;
- pelo ambiente de processamento assumido pela linguagem;
- pelo paradigma de programação principal, assumido pela linguagem.

Nesta secção e no seguimento da estratégia adoptada nas fases anteriores[63], pretende-se classificar as linguagens de programação quanto ao *paradigma principal* suportado.

Ambler, Burnnet e Zimmerman (1992, p. 28) consideram que "um paradigma de programação é uma colecção de padrões conceptuais que juntos moldam o processo de desenho e determinam a estrutura do programa". De um modo mais simples, considera-se que um paradigma de programação é uma perspectiva abstracta de uma classe de linguagens de programação que descreve um modo de solucionar problemas de programação (Hailpern 1986), isto é, um paradigma é uma determinada abordagem à programação

Considera-se que uma linguagem de programação suporta um determinado paradigma de programação, se fornece as primitivas desse paradigma, os métodos de composição necessários e a linguagem é apropriada para tornar os programas escritos neste paradigma claros (Bobrow 1984). De uma forma mais simples, Stroustrup (1988, p 10) afirma que "uma linguagem suporta um paradigma de programação se fornece facilidades que tornam conveniente (razoavelmente simples, seguro e eficiente) utilizar o paradigma".

[63] Análise dos requisitos do software e concepção do software.

Descreve-se em seguida as propostas de alguns autores sobre possíveis classificações das linguagens de programação, quanto ao paradigma suportado.

Para Watt (1990) e Placer (1991), as linguagens de programação, quanto ao paradigma suportado podem ser divididas em: imperativas, orientadas aos objectos, concorrentes, funcionais, lógicas e relacionais.

Friedman (1991), considera que as linguagens de programação podem ser classificadas em:

1. Centradas no processo;
    - Imperativas;
    - Fluxos de dados;
    - Funcionais;

2. Centradas nos dados;
    - Restrições;
    - Lógicas;
    - Objectos;
    - Bases de dados.

Segundo Hailpern (1986), existem diversos paradigmas de programação, por exemplo: orientado aos acessos, orientado às estruturas de dados, funcional, imperativo, orientado aos objectos, paralelo, orientado às regras.

Finalmente Ambler, Burnnet e Zimmerman (1992), apresentam a seguinte divisão, quanto ao paradigma suportado:

1. Operacionais;
    - *Side-effecting*;
        i. Imperativas;
        ii. Orientadas aos objectos;
    - *Non-side-effecting*;
        i. Funcionais;

2. Definicionais;
    - Lógicas;
    - Restrições;
    - Baseadas em formulários;
    - Fluxos de dados;
3. Demonstracionais.

Adopta-se a classificação proposta por Friedman (1991), pois analisa as linguagens de programação em relação à sua aproximação aos dados ou processo, devido à sua simplicidade e uma vez que cobre as linguagens de programação mais utilizadas nos meios académicos e empresariais. Considera-se então, nesta secção, que as linguagens de programação, quanto ao paradigma suportado, podem ser classificadas em: **centradas no processo** e **centradas nos dados**.

As linguagens de programação centradas no processo, preocupam-se com os detalhes de calcular a solução e com o processo de decomposição do problema (Friedman 1991) e dividem-se em **imperativas** e **funcionais**.

As linguagens de programação centradas nos dados, suportam o desenvolvimento de programas através da especificação dos dados que serão manipulados (Friedman 1991) e classificam-se em **lógicas**, **orientadas aos objectos** e **relacionais**.

Refira-se que na classificação adoptada, ignoraram-se as linguagens de programação em fase experimental ou pouco divulgadas[69]. Em resumo, utiliza-se a seguinte classificação:

PARADIGMA

Centradas no processo
    Imperativas
    Funcionais
Centradas nos dados
    Lógicas
    Orientadas aos objectos
    Relacionais

[69] Por exemplo, Forms/3 (Viehsteadt e Ambler 1992), Show and Tell (Kimura, Choi e Mack 1990) ou Metamouse (Maulsby e Witten 1989).

3.4.1. Centradas no processo

3.4.1.1. Imperativas

Grande parte das linguagens de programação mais divulgadas actualmente, suportam o paradigma de programação imperativo[70]. Este paradigma preocupa-se fundamentalmente com as variáveis requeridas, a sua associação a determinadas localizações em memória e a sequência de transformações nos valores armazenados, de modo que o estado final apresente um valor resultante correcto (Ambler, Burnnet e Zimmerman 1992).

A filosofia das linguagens baseadas no paradigma imperativo, está intimamente ligada à arquitectura das máquinas Von Newmann. Esta influência, verifica-se em três características das linguagens imperativas (Ghezzi e Jazayeri 1987):

L. Imperativas		**M. Von Newmann**
variáveis	⇨	localizações na memória
atribuir valor à variável	⇨	armazenar valor na memória
sequência de instruções	⇨	sequência de instruções

Devido a este relacionamento com a arquitectura física, em princípio as linguagens imperativas podem ser implementadas de um modo eficiente (Watt 1990)

Considera-se que este paradigma é marcado pela utilização fundamental de comandos de atribuição e de estruturas de controlo do fluxo do programa, verificando-se uma nítida distinção entre programa e dados (Friedman 1991). Os componentes principais das linguagens imperativas são, variáveis, comandos e procedimentos.

Existe um elevado número de linguagens de programação que suporta o paradigma imperativo, por exemplo: Cobol, Fortran, C, Pascal, Basic e Ada

[70] Do Latim *imperare*, que significa comandar.

3.4.1.2. Funcionais

Segundo Ambler, Burnnet e Zimmerman (1992), o paradigma funcional baseia-se no modelo matemático de composição funcional, isto é, o resultado de uma computação é a entrada para a seguinte, e assim repetidamente até que uma composição permita obter o resultado desejado. As linguagens de programação que suportam o paradigma funcional, denominam-se linguagens funcionais ou declarativas.

Nas linguagens funcionais, considera-se que um programa é uma função (ou um grupo de funções), tipicamente composta a partir de funções mais simples. Os relacionamentos entre as funções são de dois tipos, uma função pode chamar outra ou o resultado de uma função pode ser utilizado como argumento de outra. As variáveis, os comandos e os efeitos laterais[71] são excluídos e os programas são escritos apenas através de expressões, funções e declarações (Watt 1990).

Ao banir o conceito de atribuição, assegura-se que as funções não têm efeitos laterais e comportam-se como funções matemáticas. A ordem de chamada das funções deixa de interessar, invocando uma função com determinado valor produz sempre o mesmo resultado. A este facto denomina-se transparência referencial.

Os programas nas linguagens funcionais, declaram *o que* deve ser feito e não *como*[72] deve ser feito, verificando-se que estão num nível de abstracção superior ao das linguagens imperativas, tornando possível a produção de soluções mais simples para muitos problemas. Esta característica torna as linguagens funcionais vantajosas no desenvolvimento rápido de protótipos (Wilson e Clark 1993).

Normalmente, aponta-se a estas linguagens a sua ineficiência como principal desvantagem, embora seja compensada pelo seu elevado nível de abstracção e pela sua adaptação à implementação concorrente (Watt 1990) e (Ghezzy e Jazayeri 1987).

Como exemplos das linguagens funcionais, temos, o Lisp, Ml, Fp e o Logo, entre outras.

[71] *Side-effects*.
[72] Característica das linguagens imperativas.

3.4.2. Centradas nos dados

3.4.2.1. Lógicas

Segundo Ghezzi e Jazayeri (1987), o desenvolvimento de um programa utilizando o paradigma lógico, consiste na declaração dos factos e propriedades que descrevem o problema. Esta informação é utilizada pelo sistema para chegar ao objectivo[73] (inferir a solução), sem necessidade de mais contributos do programador sobre como lidar com a informação. Verifica-se que, na programação lógica a descrição do problema é realizada através de um formalismo lógico, normalmente as cláusulas Horn[74] (Ambler, Burnnet e Zimmerman 1992) e (Kowalski 1985).

Grande parte das linguagens de programação baseia-se na perspectiva de *como* as operações devem ser executadas. O paradigma lógico é orientado ao *objectivo*, ou seja, o programador define o problema a ser resolvido. Ao programador não é pedida uma solução detalhada de como o objectivo pode ser alcançado, isto é deixado ao próprio sistema (Wilson e Clark 1993).

Considera-se também que a programação lógica, se baseia na noção de que um programa implementa uma *relação*, ao contrário dos restantes paradigmas de programação que se baseiam na perspectiva de que um programa implementa um *mapeamento*. Uma vez que as relações são mais gerais que mapeamentos, a programação lógica apresenta potencialmente um nível de abstracção mais elevado do que os outros paradigmas (Watt 1990). O objectivo final da programação lógica é possibilitar a execução da especificação directamente, sem necessidade de refinamento para um nível de abstracção mais baixo.

De um modo mais preciso, considera-se que a programação lógica, consiste em (Ghezzi e Jazayeri 1987):

- Modelar a situação através da especificação de factos sobre os objectos;
- Especificar algumas regras sobre o sistema que está a ser modelado;
- Solicitar ao sistema para inferir respostas às questões, aplicando um procedimento de inferência à sua base de conhecimentos de factos e regras.

Programar consiste então em declarar correctamente todos os factos e regras

[73] *Goal*.
[74] Um subconjunto da lógica de predicados de primeira ordem.

As objecções que se colocam às linguagens de programação lógicas, são principalmente de dois tipos, a falta de transparência do código - falta uma estrutura do tipo 'módulo' que imponha forma e estrutura ao programador - e a falta de eficiência dos programas (Wilson e Clark 1993)

A programação lógica, apresenta como vantagens, a sua adaptação natural ao desenvolvimento de especificações executáveis (prototipagem rápida) e a problemas que não são facilmente expressos em termos de regras de produção, por exemplo, tradução da linguagem natural (Ambler, Burnnet e Zimmerman 1992). O Prolog[75] é a linguagem de programação mais divulgada, que suporta o paradigma lógico

3.4.2.2. Orientadas aos objectos

Segundo Thomas (1989), considera-se que as linguagens de programação orientadas aos objectos, pertencem a um novo paradigma de programação, devido fundamentalmente à encapsulação dos métodos e do estado e à programação incremental através do polimorfismo e da herança.

Considera-se que as linguagens de programação orientadas aos objectos, implementam de um modo prático os princípios difundidos pela programação estruturada, isto é, decomposição do programa, abstracção de dados e não visibilidade da informação[76]. Verifica-se assim que a abordagem (paradigma) orientada aos objectos, promove e facilita a evolução do software (Pokkunuri 1989).

Os conceitos fundamentais da programação orientada aos objectos são: classes, objectos, passagem de mensagens e herança (Watt 1990) e (Thomas 1989). Considera-se que uma *classe* faz a encapsulação das declarações de dados, denominadas variáveis de instância e das especificações das funções, chamados métodos, necessários para a definição do objecto. Um *objecto*, é uma instância de uma classe. Verifica-se que os objectos são definidos por uma hierarquia de classes, com capacidade de *herança*. Esta hierarquia permite que as variáveis e métodos de uma classe sejam herdados pelas suas subclasses, não sendo necessário codificá-los novamente.

[75] *PROgramming in LOGic*
[76] *Information hiding*.

Conclui-se então que os objectos agrupam dados e funções relacionadas, comunicando entre si através da *passagem de mensagens*. A recepção de uma mensagem por parte de um objecto implica a activação do método (operação) correspondente (Friedman 1991).

Segundo Cox (1991), os maiores benefícios da programação orientada aos objectos, são:

- elevado grau de modularidade (encapsulação);
- gestão natural da complexidade inerente aos sistemas reais;
- adaptação à reutilização de código (herança);
- diminuição efectiva dos erros em cadeia.

As desvantagens apontadas às linguagens de programação orientadas aos objectos, referem-se à ineficiência devido à junção dinâmica[77], à gestão dinâmica de memória[78] e ao próprio tempo de processamento necessário para o tratamento das mensagens que é superior ao verificado na invocação de subprogramas. Para Friedman (1991, p. 199), na programação orientada aos objectos "o ênfase é a grande produtividade dos programadores à custa (por vezes) da eficiência da máquina".

As linguagens Smalltalk e Eiffel, baseiam-se no paradigma orientado aos objectos Existem outras linguagens que suportam este paradigma, mas cuja base é o paradigma imperativo, por exemplo, C++, ObjectPascal, VisualBasic, etc (Wilson e Clark 1993).

3.4.2.3. Relacionais

Segundo Placer (1991) o paradigma relacional baseia-se no 'mundo' das relações, que podem ser pensadas como tabelas. Os operadores fornecidos pelas linguagens de programação (linguagens de quarta geração[79]) que suportam o paradigma relacional operam sobre tabelas existentes com o objectivo de criar novas tabelas e seleccionar a informação pretendida.

[77] *Dynamic binding*.
[78] *Garbage collection*.
[79] *4th Generation Languages* (4GL).

Friedman (1991), considera que uma linguagem de quarta geração deve apresentar as seguintes caracteristicas:

- fácil aprendizagem;
- centrada nos dados (base de dados ou base de conhecimento);
- maximizar a produtividade humana;
- essencialmente não procedimental;
- fácil manutenção;

Verifica-se que uma linguagem de quarta geração selecciona a informação desejada, através da especificação das propriedades lógicas que caracterizam tal informação. Não é dito nada sobre como aceder a essa informação na base de dados (Ghezzi e Jazayeri 1986).

As operações principais suportadas pelas linguagens de quarta geração são, a *definição de dados*, isto é a especificação da estrutura das tabelas e das restrições necessárias; a *manipulação de dados*, que inclui a recuperação e manutenção de dados; a *apresentação de dados*, que se refere a técnicas para impressão de relatórios ou a visualização de dados (Nijssen e Halpin 1989).

O SQL[80] é o standard de *facto* das linguagens de quarta geração, embora existam outras linguagens, por exemplo, o QBE[81] ou o Intellect (Friedman 1991).

Para McDermid (1991) as linguagens de quarta geração, embora apresentem os requisitos necessários para seleccionar a informação da base de dados, não são normalmente suficientes para o desenvolvimento software, pois não apresentam uma funcionalidade completa, não se adaptam a determinados tipos de utilizadores, verifica-se que o acesso a outras capacidades é inexistente (por exemplo gráficos) e torna-se bastante ineficiente em grandes aplicações.

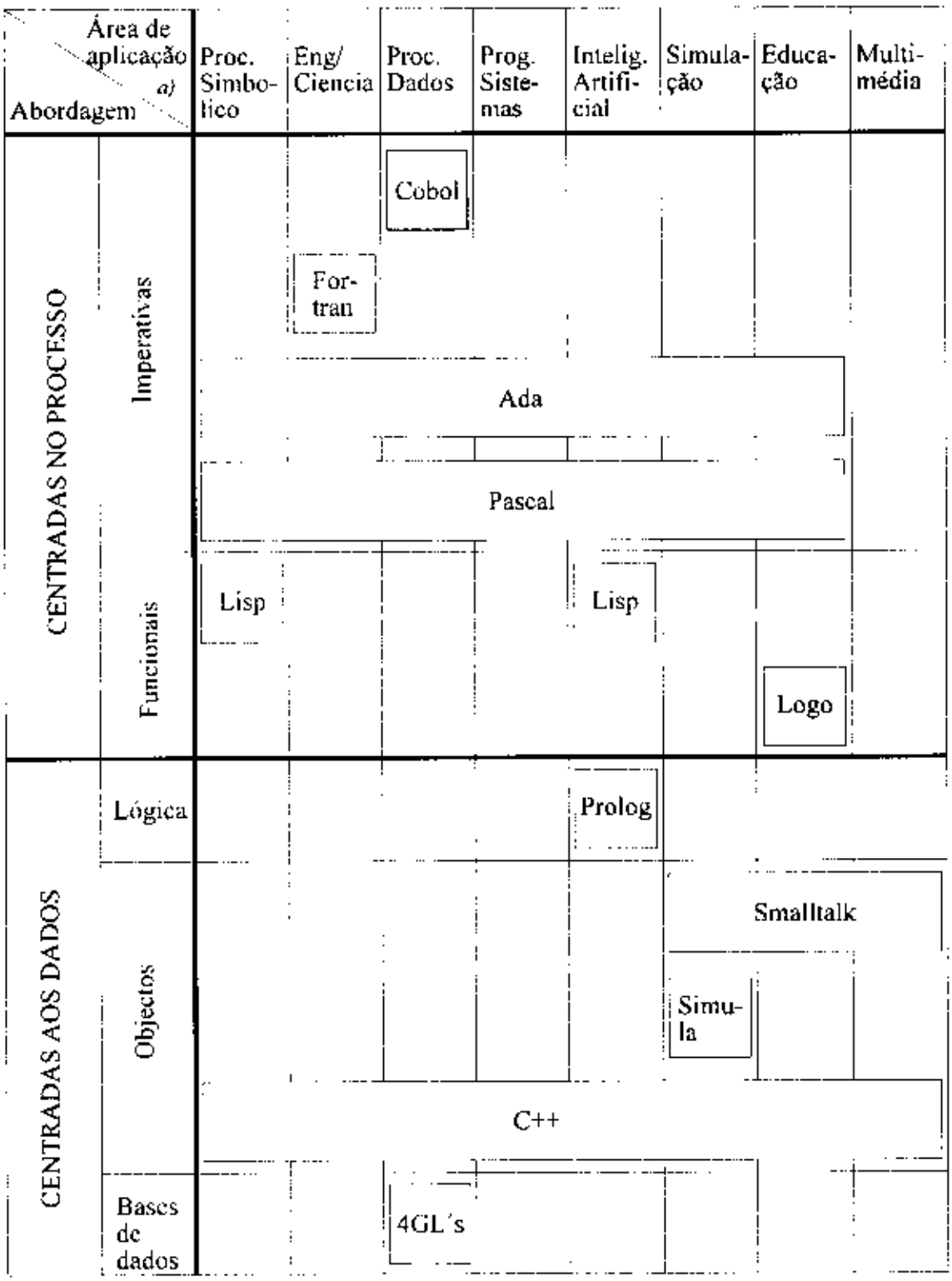
Normalmente o sistema de gestão de base de dados (SGBD), adopta a abordagem de fornecer um mecanismo através do qual, uma linguagem de alto nível (normalmente C, Pascal ou Cobol) é aumentada para suportar as primitivas das linguagens de quarta geração (McDermid 1991).

[80] *Structured Query Language*.
[81] *Query by Example*.

3.4.3. Classificação (quanto à abordagem) das linguagens de programação

Abordagem		Exemplos
Centrada no processo	Imperativas	Cobol, Pascal, C, Ada, Fortran, Basic
	Funcionais	Lisp, Ml, Fp, Logo
Centrada nos dados	Lógica	Prolog
	O. Objectos	Smalltalk, Eiffel, C++, ObjectPascal
	Relacionais	SQL, QBE, Intellect (4GL)

3.4.4. Classificação (quanto à abordagem e áreas de aplicação) das linguagens de programação



a) Adaptado de Friedman (1991)

4. ANÁLISE DAS TRANSFORMAÇÕES

Definiu-se[81] uma *transformação* (TS), como uma 'passagem' entre duas representações. Nesta secção, pretende-se analisar e descrever algumas transformações mais divulgadas.

Refira-se que a maior parte destas transformações existem no contexto de um método de desenvolvimento de software, caracterizando-se normalmente, pelo grau de simplicidade de aplicação e formalismo inereme (Moynihan 1993).

Ao analisar uma transformação sob a perspectiva das representações inicial/final e sob a perspectiva da evolução no processo de desenvolvimento de software, verifica-se a conveniência de dividir as transformações em três grupos, tais como:

- *transformação completa*, caracterizada por técnicas de representação diferentes no inicio e fim e por uma mudança na fase do processo de desenvolvimento de software,

- *transformação parcial*, verifica-se uma evolução no processo de desenvolvimento de software, contudo a representação inicial/final é a mesma;

- *transformação equivalente*, consiste numa 'passagem' entre duas técnicas de representação similares, na mesma fase do processo de desenvolvimento de software e apresentando a mesma perspectiva (conteúdo). Verifica-se contudo uma mudança de representação.

[81] Ver secção 2.7.2.

No âmbito desta dissertação, considera-se que as transformações completas apresentam um maior interesse, uma vez que são as mais divulgadas, são parte fundamental da maioria dos métodos de desenvolvimento de software e contribuem de forma decisiva para a obtenção do software final.

Com este objectivo, apresenta-se em seguida uma descrição global das transformações parciais e equivalentes e uma descrição detalhada das transformações completas. Esta descrição detalhada inclui uma breve descrição de alguns exemplos de transformações, definindo-se as representações inicial/final[82] da transformação quanto à forma, conteúdo e fase do processo de desenvolvimento de software coberta.

Ao longo desta dissertação, apresentaram-se alguns exemplos de **transformações parciais** realizadas sobre determinadas representações:

- Diagrama de entidades e relacionamentos[83];
- Diagrama NIAM[84];
- Técnicas formais[85].

Existem outros casos, ligados à definição da fronteira entre a análise dos requisitos do software e a concepção do software. Note-se que uma representação utilizada na definição do problema (especificação dos requisitos), é por vezes utilizada também, para conceber a solução de alto nível (especificação do software). Estas transformações são normalmente 'informais', isto é, dependem essencialmente da experiência do analista e da estruturação do próprio sistema real.

Este facto (difícil distinção entre a análise dos requisitos e concepção) é mais marcante nas técnicas de representação orientadas aos objectos, considerando-se por isso que o diagrama de Coad, o DFDE e o diagrama de Rumbaugh podem sofrer transformações parciais. Verifica-se contudo que os diagramas de fluxos de dados e os diagramas de especificação do sistema são também utilizados para a concepção de alto nível.

[82] Utiliza-se o modelo de representações/transformações (secção 2.7.2.) e as classificações das técnicas de representação adoptadas (secções 3.2.3, 3.3.3 e 3.4.3).
[83] Ver secção 3.3.2.1.2.
[84] Ver secção 3.3.2.1.2.
[85] Ver secção 3.3.1.

Verifica-se que existem dois tipos de **transformações equivalentes**.

1. entre *representações similares*, considera-se que as técnicas de representação inicial/final, pertencem a uma mesma família. Diz-se que duas representações pertencem à mesma família, quando apresentam a mesma informação e utilizam conceitos similares mas a notação é diferente (Hull e O'Donoghue 1993);
2. entre *representações diferentes*, verifica-se que a fase do processo de desenvolvimento de software e o conteúdo são o mesmo, contudo os conceitos essenciais e a notação são diferentes.

A transformação entre duas representações da mesma família é normalmente simples e directa. Como exemplos deste tipo de transformação temos:

- DCVE $\xleftarrow{TS}$ DTE;
- DFD $\xleftarrow{TS}$ DES[36].

Refira-se que estas transformações são utilizadas com objectivos específicos, por exemplo, uma mudança no método de desenvolvimento utilizado, maior adaptação de uma representação para determinado tipo de problemas e melhor interface com o cliente da representação (utilizador, analista, projectista ou programador).

O segundo tipo de transformações equivalentes (entre representações diferentes), apresenta-se normalmente mais complexa, embora vários autores, nomeadamente, Kemmerer (1990), Ward (1989) e France (1992) apresentem diversas vantagens na sua utilização. Os exemplos mais comuns desta transformação, são:

- DER $\xrightarrow{TS}$ Diagrama de Coad;
- DFD $\xrightarrow{TS}$ Métodos formais;
- DFD $\xrightarrow{TS}$ DFDE.

Verifica-se que o objectivo principal desta transformação é utilizar uma transformação considerada 'madura', inicialmente[37], e através desta transformação obter as vantagens da abordagem orientada aos objectos ou dos métodos formais.

[36] Diagrama de Especificação do Sistema.
[37] Melhor suporte de ferramentas CASE e maior habituação dos utilizadores.

4.1. Transformações completas

Verifica-se que existem poucas transformações bem caracterizadas entre técnicas de representação. Nesta secção descrevem-se algumas transformações bastante utilizadas na prática.

- **Diagrama de Fluxo de Dados $\xrightarrow{TS}$ Diagrama de Estrutura.**

Esta transformação incluida na concepção estruturada, é obtida através de duas estratégias principais: análise de transformação e análise de transacção. As principais contribuições para a caracterização desta transformação devem-se a Yourdon e Constantine (1978) e Page-Jones (1988).

Segundo Pressman (1992) e Martin e McClure (1991) a transição de um DFD para um diagrama de estrutura é realizada através de um processo de quatro passos:

1. Verificar qual o tipo de fluxo de informação característico do DFD (centro de transformação, centro de transacção, ou uma combinação de ambos);
2. Definir as fronteiras dos fluxos;
3. Desenvolver o esboço inicial do diagrama de estrutura, utilizando os passos anteriores;
4. Avaliar e refinar o diagrama de estrutura, utilizando diversos critérios de qualidade, por exemplo, a coesão, acoplamento, *factoring*, balanceamento, etc

Page-Jones (1988) afirma que a análise de transformação é a principal estratégia para converter um diagrama de fluxo de dados num diagrama de estrutura e define-a como um modelo de fluxo de informações, utilizado na concepção de um programa através da identificação dos componentes funcionais e das entradas e saídas de alto nível para estes componentes.

Na análise de transformação o segmento de entrada no DFD, que contém os processos que transformam os dados físicos de entrada num formato lógico, denomina-se *ramo aferente*. A parte de saída que transforma os dados lógicos em dados físicos designa-se *ramo eferente*. Finalmente os processos que realizam o processamento lógico da informação denominam-se *centro de transformação* (Martin e McClure 1991).

A análise de transacção é uma estratégia alternativa para desenvolver um diagrama de estrutura e tem duas funções principais. dividir um DFD de elevada complexidade ou combinar diagramas de estrutura individuais de transacções separadas num único diagrama de estrutura (Page-Jones 1988).

Considera-se que uma transacção é um estímulo para um sistema que possui um conjunto de actividades a serem realizadas internamente. O diagrama de estrutura obtido através da aplicação desta estratégia, apresenta no topo o módulo *centro de transacção* e no nível seguinte existe um módulo de transacção para cada tipo de actividade.

Sobre esta transformação Champeaux e Faure (1992) afirmam que, apesar das estratégias aconselhadas[88], verifica-se que é uma tarefa complexa, pois o mapeamento não é verdadeiramente isomórfico. Page-Jones (1988) refere também que esta transformação é apenas um estratégia e não um algoritmo, ou seja, não existe a certeza de obtermos um resultado correcto.

Como se apresentou, esta transformação permite converter um diagrama de fluxo de dados num diagrama de estrutura. Em relação à classificação das técnicas de representação definida, verifica-se.

	Diagrama de Fluxo de Dados ⇨	*Diagrama de Estrutura*
Fase do PDS[89]:	Análise dos requisitos do software	Concepção do software
Forma:	Não formal e diagramática	Não formal
Conteúdo:	Orientada aos processos	Arquitectura dos módulos

- **Diagrama de Estrutura de Dados** $\xrightarrow{TS}$ **Diagrama de Estrutura do Processo**

Esta transformação permite conceber o modelo lógico do programa utilizando as estruturas de dados definidas na especificação dos requisitos. O princípio fundamental desta abordagem considera que a estrutura necessária do processo tem que incorporar tanto as estruturas de dados de entrada como de saída (Skidmore, Farmer e Mills 1992).

[88] Análise de transformação e análise de transacção.
[89] Processo de Desenvolvimento de Software.

A definição inicial desta transformação deve-se ao método JSD (Jackson 1983) verificando-se que a fase de concepção do SSADM[90] (Downs, Clare e Coe 1992) se baseia também neste processo.

A similaridade nos requisitos de processamento entre duas estruturas de dados (entrada/saída), designa-se correspondência de dados e existe no nível dos componentes. Davies e Layzell (1993) consideram que dois componentes de uma estrutura de dados *correspondem*, se exibem:

- um relacionamento funcional;
- um relacionamento um para um;
- uma equivalência em relação à ordem temporal;
- uma equivalência hierárquica

Ao mecanismo de geração da estrutura do processo a partir das correspondências de dados, ou seja à transformação em análise, denomina-se **geração da estrutura do processo** e consiste:

1. num conjunto de regras de correspondência estrutural;
2. no caso de existir *colisão de estrutura*[91] (colisão de ordem, colisão de fronteira ou colisão entre folhas), nas possíveis opções a tomar.

Refira-se como conclusão que, o conceito de correspondência é difícil de visualizar, verificando-se que esta transformação é um processo intuitivo, muito dependente do conhecimento inerente do sistema, apresentado pelo projectista[92] (Davies e Layzell 1993). Situando esta transformação em relação à classificação das representações, obtém-se:

	D. de Estrutura dos Dados ⇨	*D. Estrutura do Processo*
Fase do PDS:	Análise dos requisitos do software	Concepção do software
Forma:	Não formal e diagramática	Não formal
Conteúdo:	Orientada aos dados	Arquitectura dos módulos

[90] Ver secção 3.3.2.1.1.
[91] Ver secção 3.3.2.1.1.
[92] *Designer*.

- **Diagrama NIAM $\xrightarrow{TS}$ Modelo Relacional**

Considera-se que esta transformação, fornece um procedimento sistemático que permite agrupar os tipos de factos de um esquema conceptual, num conjunto de tabelas de um esquema relacional. Esta transformação denomina-se algoritmo da Forma Normal Óptima[93] (FNO) e encontra-se definida no método NIAM (Nijssen e Halpin 1989).

Este algoritmo, tem como objectivo produzir estruturas de dados simples, seguras e eficientes e verifica-se que as tabelas desenvolvidas satisfazem três critérios: **i.** inexistência de atributos repetidos; **ii.** inexistência de problemas de redundância ou actualização, **iii.** menor número de tabelas satisfazendo os pontos i e ii.

O algoritmo FNO, apresenta as seguintes estratégias de agrupamento (Nijssen e Halpin 1989):

1. Para cada tipo de facto sem uma chave simples, criar uma tabela separada.
2. Agrupar tipos de facto com uma chave simples, ligada a um tipo de objecto comum, na mesma tabela.
3. Para cada tipo de facto sobrante criar uma tabela separada.

Nijssen e Halpin (1989, p.254), observam a simplicidade do algoritmo e afirmam que a transformação "permite gerar tabelas na quinta forma normal, (..) embora possam ser optimizadas em determinadas aplicações" Em relação à classificação das técnicas de representação definida, verifica-se:

	Diagrama NIAM	⇨	*Modelo Relacional*
Fase do PDS:	Concepção do software		Concepção do software
Forma:	Não formal		Não formal
Conteúdo:	Estruturas de dados (lógicas)		Estruturas de dados (fís.)

- **Diagrama de Entidades e Relacionamentos $\xrightarrow{TS}$ Modelo Relacional**

Verifica-se que esta transformação, permite passar um modelo lógico de dados para um modelo físico de dados, sobressaindo a sua grande simplicidade de aplicação.

[93] *Optimal Normal Form* (ONF).

A transformação é realizada através de duas tarefas bem definidas (Ullman 1982) e (Elmasri e Navathe 1989):

1. Cada *entidade* origina uma tabela, utilizando-se os atributos e chave da entidade como atributos e chave da relação.
2. Um relacionamentos *1:M* no DER, é transformado numa chave externa (chave candidata da entidade que participa no relacionamento, com cardinalidade *um*) e acrescentado na entidade com cardinalidade *muitos*.

Refira-se que o diagrama de entidades e relacionamentos, considerado como entrada para esta transformação, deve estar normalizado e apresentar outros aspectos relevantes (desempenho, segurança, tempos de resposta, etc) convenientemente concebidos. Refira-se também a importância desta transformação, devido à sua utilização em diversos métodos de desenvolvimento de software, à sua facilidade de aplicação e à forte aproximação entre o modelo relacional e os sistemas de gestão de bases de dados (SGBD relacional).

Classificando esta transformação em relação às representações de entrada e saída, obtém-se:

	DER	⇨	*Modelo Relacional*
Fase do PDS:	Concepção do software		Concepção do software
Forma:	Não formal		Não formal
Conteúdo:	Estruturas de dados (lógicas)		Estruturas de dados (fis)

- **Diagrama de Estrutura** $\xrightarrow{TS}$ **Linguagens Imperativas**
  L. de Concepção de Programas

Verifica-se que as linguagens imperativas, são também denominadas linguagens algorítmicas, porque facilitam a construção de programas como algoritmos ou como sequências de instruções para computação (Friedman 1991).

Adicionalmente o subprograma é parte integrante da abordagem imperativa, definindo-se como um bloco de instruções com as seguintes características: possui um nome; é relativamente independente; executa tarefas específicas. Existem diversos tipos de subprogramas com diferentes designações, por exemplo, parágrafo, procedimento, subrotina ou função.

Considerou-se que o diagrama de estrutura apresenta a estrutura hierárquica de módulos e os parâmetros passados entre eles. A linguagem de concepção de programas define os detalhes dos procedimentos de um módulo.

A transformação do conjunto, diagrama de estrutura e linguagem de concepção de programas para uma linguagem imperativa é apenas uma tradução da solução concebida, para as primitivas da linguagem de programação. Normalmente esta tradução é pouco complexa, uma vez que as linguagens imperativas possuem as construções necessárias para uma transformação directa.

	i *Diagrama de Estrutura* ⇨	*L. Imperativas*
	ii. *L. Concepção de Programas*	
Fase do PDS:	Concepção do software	Implementação do s/w
Forma:	Não formais	-
Conteúdo:	i. Arquitectura dos módulos	Imperativo
	ii. Detalhes dos procedimentos	

- **Modelo Relacional** $\xrightarrow{TS}$ **SGBD Relacional**

Segundo McDermid (1991), considera-se que um sistema de gestão de bases de dados, possui três componentes principais:

1. um processador de querys, que aceita pedidos de manipulação de dados dos utilizadores, convertendo-os para uma forma executável;
2. um gestor de transacções, que gere a execução dos pedidos de manipulação da base de dados;
3. um conjunto de ferramentas que fornecem a possibilidade de executar diversas tarefas, tais como: criação da base de dados, re-estruturação da base de dados, avaliação do desempenho e desenvolvimento de aplicações.

A transformação entre o modelo relacional e um SGBD relacional, consiste apenas na tradução das tabelas para a linguagem especifica definida pelo SGBD. Utiliza-se as facilidades propostas pelo SGBD para implementar o modelo relacional.

Verifica-se que esta transformação é normalmente simples, pouco morosa e a implementação resultante reflete a solução concebida (não existe espaço para alterações ligadas à filosofia de programação).

	Modelo Relacional	⇨	*SGBD Relacional*
Fase do PDS:	Concepção do software		Implementação do s/w
Forma:	Não formal		-
Conteúdo:	Estruturas de dados (físicas)		Relacional

4.2. Discussão sobre transformações

A análise comparativa da classificação das técnicas de representação utilizada na análise dos requisitos do software[94] e na concepção do software[95], permite distinguir duas grandes estratégias no desenvolvimento de software: abordagem convencional e abordagem orientada aos objectos.

A divisão anterior, baseia-se na observação das técnicas de representação em relação ao seu conteúdo. Refira-se contudo que na abordagem convencional, existe uma 'forte' distinção entre o desenvolvimento formal e não formal. Considera-se então que existem três estratégias principais utilizadas no processo de desenvolvimento de software e suportadas por representações e transformações apropriadas:

- Abordagem convencional, não formal,
- Abordagem convencional, formal;
- Abordagem orientada aos objectos.

Para a caracterização das estratégias de desenvolvimento de software, utiliza-se a noção de *transformação homogénea*, isto é, uma 'passagem' entre duas representações que utilizam conceitos semelhantes e *transformação não homogénea*, uma transformação entre representações com conceitos distintos. As transformações homogéneas são normalmente mais simples e naturais

[94] Ver secção 3.2.3.
[95] Ver secção 3.3.3.

Na figura 22, analisa-se as transformações apresentadas na secção anterior, na perspectiva da sua homogeneidade e da sua integração nas três estratégias de desenvolvimento propostas.

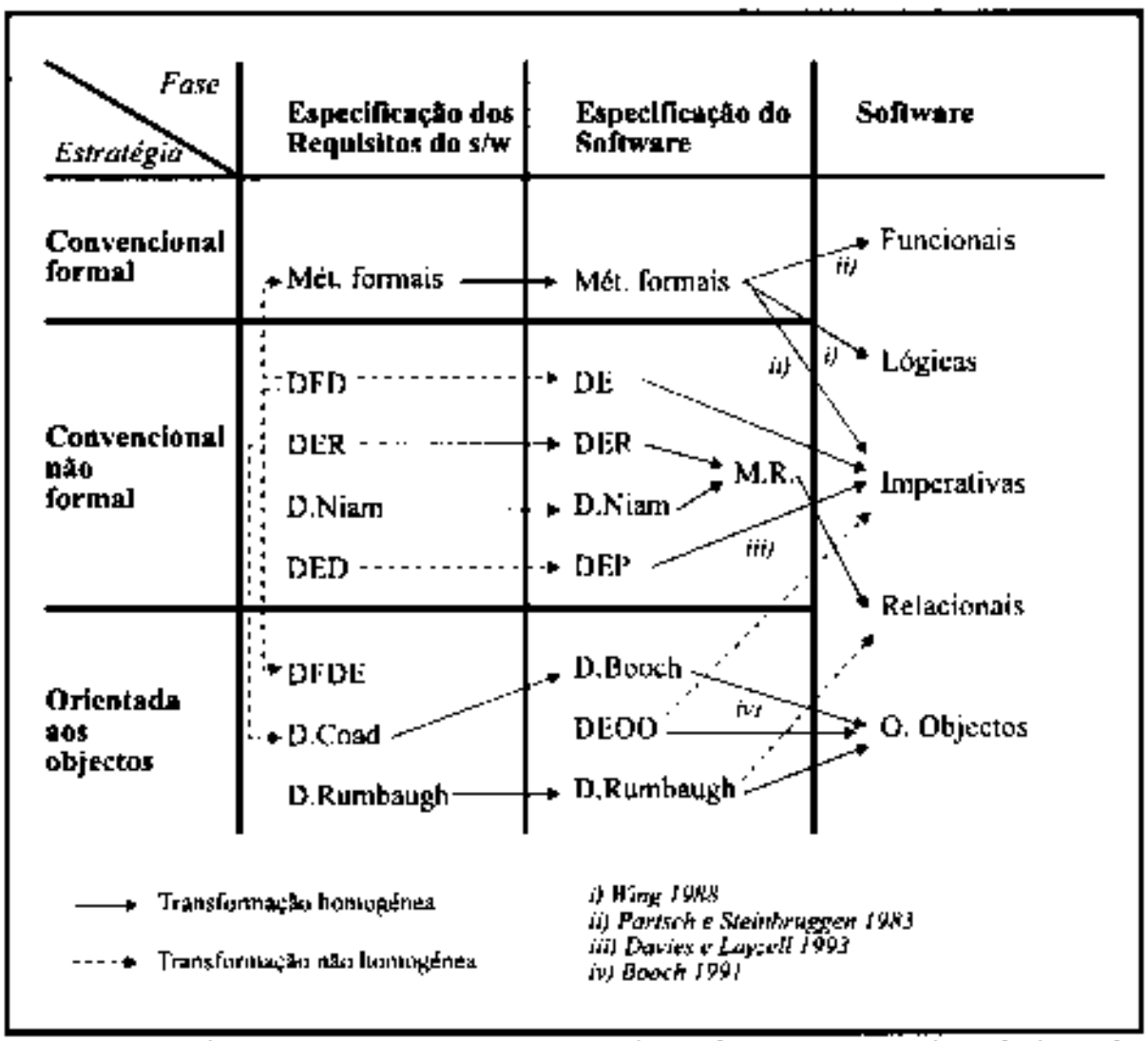


Figura 22 - Transformações entre representações características das principais estratégias de desenvolvimento de software

A abordagem convencional e não formal, caracteriza-se pela utilização de diversas representações para especificar os requisitos do software, cobrindo a perspectiva dos processos, dados e comportamento. Em seguida, utiliza-se um *novo* conjunto de técnicas de representação para especificar o software, apresentando a arquitectura dos módulos, os detalhes dos procedimentos e as estruturas de dados As transformações entre estes dois conjuntos de representações consideram-se **não homogéneas**

As representações utilizadas na especificação do software, adaptam-se à implementação em linguagens imperativas ou relacionais, verificando-se que existe uma transformação **homogénea** (Henderson-Sellers e Edwards 1990).

A abordagem convencional e não formal, revela-se bastante forte em alguns aspectos, nomeadamente:

- grande maturidade;
- fácil interface com os utilizadores não informáticos;
- bom suporte de ferramentas CASE,
- ligação homogénea às linguagens imperativas e relacionais, os tipos mais divulgados de linguagens de programação.

As suas principais desvantagens, devem-se: a) ao facto da transformação entre a especificação dos requisitos e a especificação do software ser não homogénea, b) não se adaptar à implementação orientada aos objectos e c) o seu carácter não formal.

A abordagem convencional e formal, caracteriza-se pela utilização de apenas uma técnica de representação durante todo o processo de desenvolvimento de software. Através do refinamento (reificação) a especificação inicial é sucessivamente transformada até atingir a implementação. Considera-se que as transformações neste processo são **homogéneas**. Como já foi referido[96], a implementação tem que incluir uma descrição detalhada dos dados utilizados pelo software e uma descrição detalhada dos algoritmos utilizados para executar as operações.

Verifica-se então que a abordagem convencional e formal, se adapta às linguagens imperativas, funcionais e lógicas, uma vez que o modo de reificar depende da técnica de representação final que se pretende utilizar.

Sommerville (1992), apresenta algumas vantagens ligadas à utilização dos métodos formais: 1) redução esperada de erros e omissões; 2) possibilidade de analisar as representações intermédias em termos matemáticos e 3) boa adaptação ao processamento

[96] Ver secção 3.3.1.

automático. No que se refere às desvantagens, o mesmo autor apresenta: falta de formação dos engenheiros de software nesta área; dificuldade da sua utilização em alguns aspectos específicos (por exemplo, restrições temporais) e falta de ferramentas de apoio.

Hall (1990, p.11) resume o estado actual do desenvolvimento formal, afirmando que, "Os métodos formais são controversos. Os seus defensores afirmam que eles podem revolucionar o desenvolvimento de software. Os detractores afirmam que são realmente complexos. Entretanto, para a maioria das pessoas, os métodos formais são tão desconhecidos que se torna difícil julgar os argumentos"

A abordagem orientada aos objectos, caracteriza-se pela utilização de várias representações que permitem especificar os requisitos do software (domínio-problema) e especificar o software (domínio-solução), modelando: i) serviços; ii) objectos, atributos e relacionamentos; iii) comunicação e temporização / controlo. Refira-se que todas as representações utilizadas no desenvolvimento orientado aos objectos são uniformes em termos conceituais, isto é, o objecto é a unidade central de qualquer representação. Verifica-se então que qualquer transformação entre representações é **homogénea**.

O desenvolvimento orientado aos objectos adapta-se à implementação em linguagens de programação orientadas aos objectos, verificando-se a manutenção de conceitos e considerando-se então que existe uma transformação **homogénea**

As maiores vantagens da abordagem orientada aos objectos são: a) inclusão do conceito de reutilização; b) aproximação real entre as diversas representações, o que implica transformações normalmente simples; c) clareza da documentação resultante (Capretz e Lee 1993) e (Henderson-Sellers e Edwards 1990) As maiores dificuldades, verificam-se nos seguintes aspectos: a) pouca maturidade dos métodos orientados aos objectos; b) falta de mecanismos (decomposição) de tratamento de sistemas complexos; c) o conceito de reutilização não é convenientemente tratado pela maioria dos métodos de desenvolvimento orientados aos objectos (Fichman e Kemerer 1992) e (Capretz e Lee 1993).

Na figura 22, observam-se dois aspectos a salientar, no contexto das transformações entre representações, utilizadas em estratégias de desenvolvimento de software diferentes:

1. A existência de *transformações não homogéneas* entre as abordagens convencionais (não formal para formal), na fase de análise dos requisitos do software.

2. A existência de *transformações não homogéneas* entre a abordagem convencional não formal e a abordagem orientada aos objectos. A transformação verifica-se entre representações características da análise dos requisitos do software.

Booch (1991), aconselha a utilização do método JSD, para a especificação dos requisitos do software e para ligação ao método concepção orientada aos objectos por si apresentado. Verifica-se então a utilização de *transformações não homogéneas*, entre representações características da abordagem convencional não formal e representações orientadas aos objectos, nas fases de análise dos requisitos do software e concepção do software respectivamente.

5. CONCLUSÃO

Pretendeu-se com este trabalho, analisar o desenvolvimento de software na perspectiva das representações e transformações entre representações. Para tal foi desenvolvida uma revisão de representações e transformações, estruturando-se nesta secção a informação mais interessante revista anteriormente, de modo a desenvolver algumas conclusões sobre os caminhos mais convenientes para o desenvolvimento de software.

Para o âmbito deste trabalho, seleccionou-se o modelo de representações / transformações do processo de desenvolvimento de software, devido à sua adaptação à seguinte perspectiva: *constatação de que o desenvolvimento de software é um processo de construção, transformação e modificação de artefactos linguísticos abstractos*. Esta abordagem permitiu analisar o processo de desenvolvimento de software de uma forma uniforme, ou seja, um conjunto de representações suportadas por técnicas de representação e as transformações que permitem a 'passagem' entre duas representações.

As técnicas de especificação dos requisitos do software foram classificadas quanto à forma em: formais ou não formais e quanto ao conteúdo em: orientadas aos processos, orientadas aos dados e orientadas ao comportamento. Considerou-se que a orientação aos objectos inclui as três perspectivas anteriores, embora utilize designações diferentes, isto é, serviços (processos), objectos, atributos e relacionamentos (dados) e comunicação e temporização / controlo (comportamento).

No que se refere às técnicas de especificação do software, dividiram-se em formais e não formais, quanto à forma e classificaram-se em relação ao conteúdo, como adoptando uma abordagem aos objectos ou uma abordagem convencional. A abordagem aos objectos, tal

como nas técnicas de especificação dos requisitos, cobre as perspectivas dos: i) serviços; ii) objectos, atributos e relacionamentos e iii) comunicação e temporização / controlo. Na abordagem convencional, as representações modelam a arquitectura dos módulos, as estruturas de dados ou os detalhes dos procedimentos.

Finalmente, apresentou-se uma classificação das técnicas de representação (linguagens de programação), utilizadas na implementação do software. Nesta divisão não foram consideradas as linguagens de programação pouco utilizadas no desenvolvimento de software ou ainda em fase experimental. A classificação proposta baseou-se no paradigma suportado, dividindo-se inicialmente em: centradas no processo e centradas nos dados. As linguagens de programação centradas no processo foram classificadas em imperativas e funcionais e as centradas nos dados em lógicas, orientadas aos objectos e relacionais.

Apresentou-se também uma divisão em áreas de aplicação, classificando-se em seguida diversas linguagens de programação. Pretende-se com esta divisão, obter conclusões sobre o relacionamento entre o tipo de software e a melhor abordagem ao processo de desenvolvimento de software.

Refira-se que as diversas classificações propostas foram consolidadas através da sua experimentação em várias técnicas de representação e posteriormente utilizadas para clarificar as transformações analisadas.

Após a revisão e classificação das técnicas de representação, analisou-se as transformações entre representações. Considerou-se três tipos de transformações principais: completas, parciais e equivalentes. Deu-se maior ênfase às transformações completas, pois são as mais utilizadas nos métodos de desenvolvimento de software e caracterizam-se por técnicas de representações diferentes no início / fim e por uma evolução no processo de desenvolvimento de software.

Da análise comparativa da classificação das técnicas de representação utilizada na especificação dos requisitos do software e na especificação do software, distinguiram-se três grandes estratégias de desenvolvimento de software: i) abordagem convencional, não formal; ii) abordagem convencional, formal; iii) abordagem orientada aos objectos.

Utilizou-se em seguida os conceitos de transformação homogénea e não homogénea para caracterizar a possível passagem entre representações características das três estratégias

de desenvolvimento de software. A figura 23 apresenta o desenvolvimento de software na perspectiva das abordagens propostas, dos paradigmas de programação e dos tipos de transformações (homogéneas ou não homogéneas) verificadas entre representações. Refira-se que a figura 23, foi desenvolvida através da generalização das transformações apresentadas na figura 22.

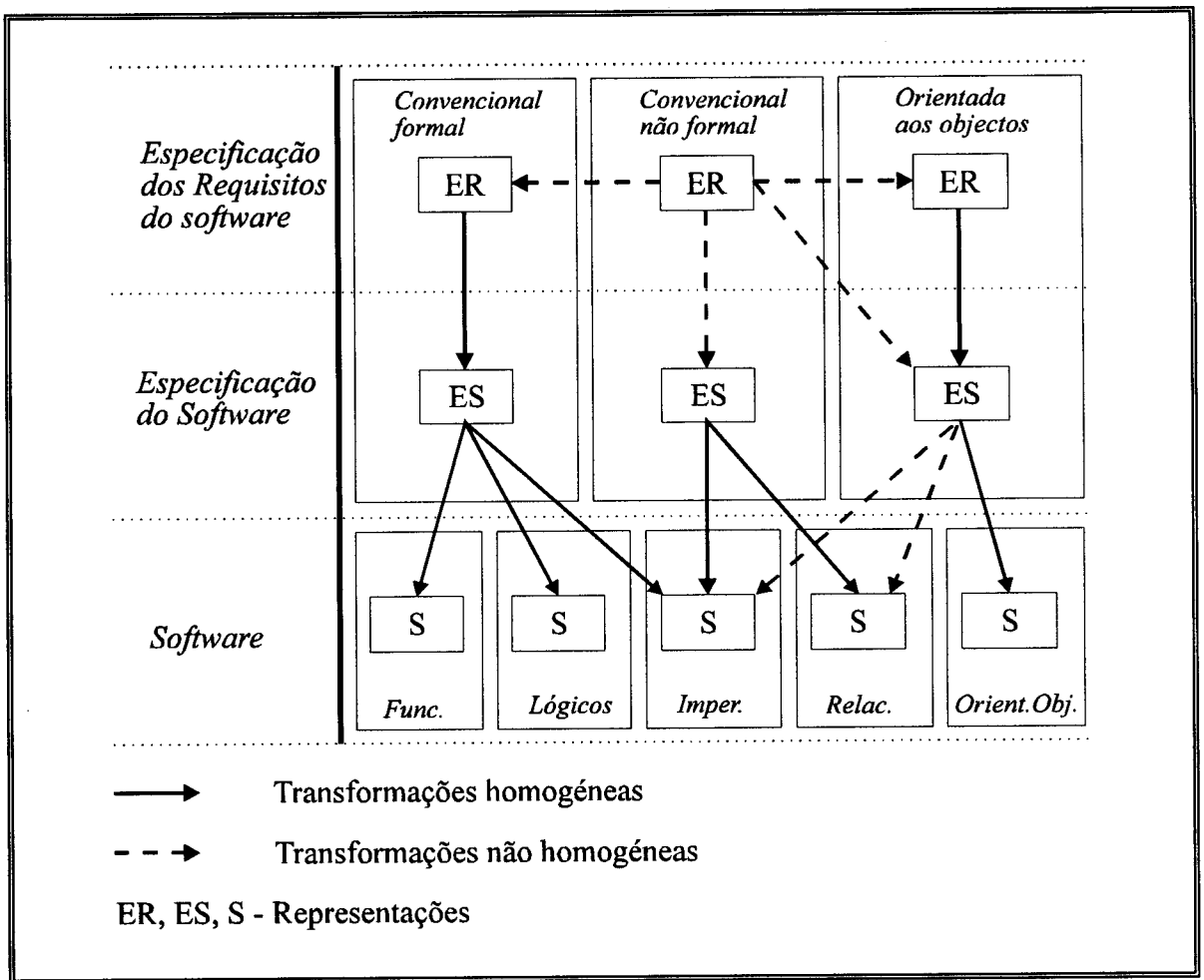


Figura 23 - Transformações entre representações nas três estratégias de desenvolvimento de software propostas.

A análise da figura 23, permite validar algumas conclusões interessantes. Em primeiro lugar, verifica-se que tanto a abordagem orientada aos objectos como a abordagem formal apresentam transformações homogéneas em todo o processo. A abordagem não formal evolui através de transformações não homogéneas entre a especificação de requisitos e a especificação do software.

Considerou-se que as transformações homogéneas são mais simples e directas do que as transformações não homogéneas, verificando-se então que a abordagem não formal

apresenta uma maior complexidade nas transformações necessárias à evolução do processo, em relação às restantes estratégias. Conclui-se que: *globalmente, as abordagens convencional formal e orientada aos objectos devem ser preferidas em relação à abordagem convencional, não formal.*

Em segundo lugar verifica-se uma preferência efectiva pela abordagem convencional não formal para a especificação dos requisitos do software. A figura 24, analisa a especificação dos requisitos do software em relação à sua aproximação ao cliente ou projectista (*designer*), observando-se que a abordagem convencional não formal está mais próxima do cliente, isto é, existe uma maior facilidade de compreensão destas representações por parte dos "leigos". Nota-se também a forte aproximação das representações formais à concepção e naturalmente uma certa dificuldade de interacção com o cliente.

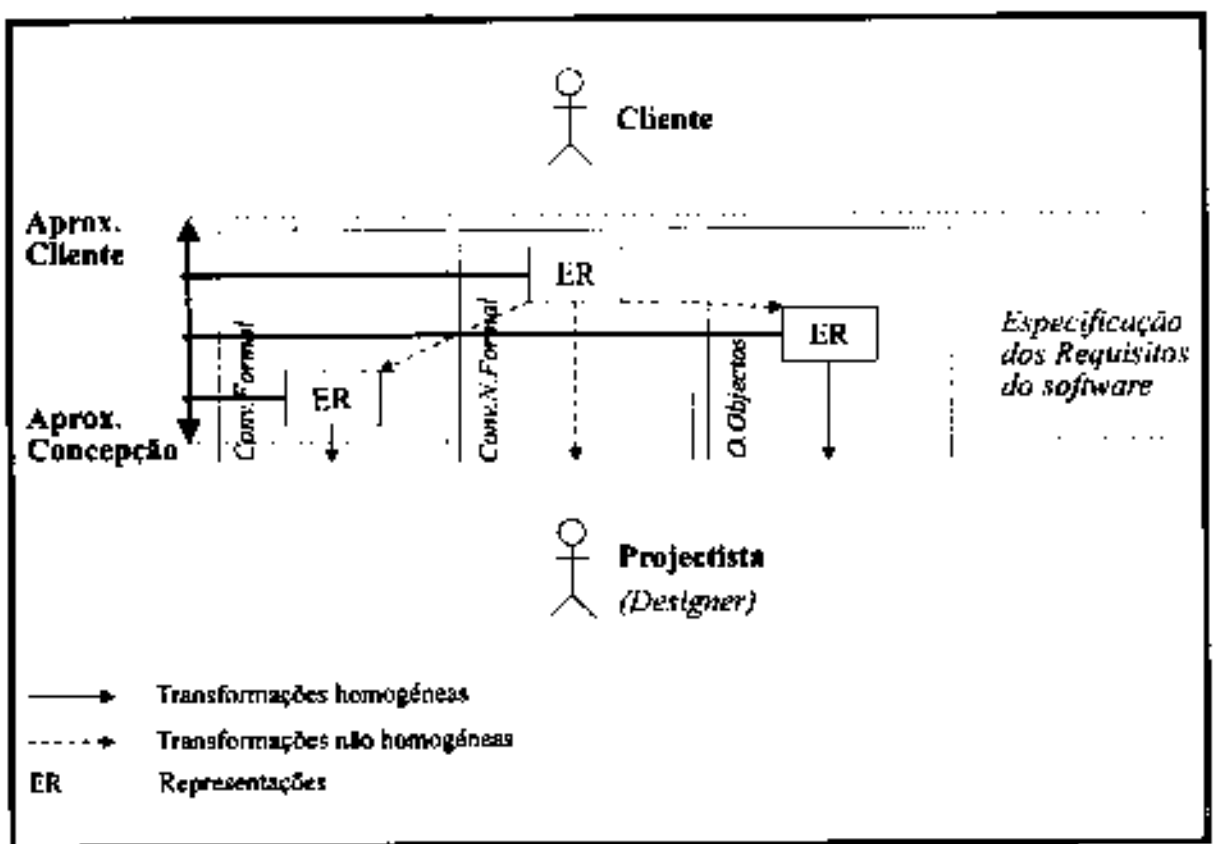


Figura 24 - Aproximação ao Cliente / Projectista na especificação dos requisitos.

Referiu-se a grande maturidade dos métodos de desenvolvimento de software baseados na estratégia não formal, sobretudo no que se refere às técnicas de especificação dos requisitos do software. Este facto, aliado aos grandes investimentos normalmente realizados pelas organizações em ferramentas de suporte às representações não formais e a sua forte

aproximação ao cliente, torna a abordagem não formal preferida pelo menos na análise dos requisitos do software. É importante contudo salientar que existem transformações não homogéneas entre as técnicas de especificação dos requisitos do software características da abordagem não formal e as técnicas de representação utilizadas na estratégia formal e orientada aos objectos. Considera-se então que, *no contexto actual justifica-se a selecção da abordagem não formal para iniciar o desenvolvimento do software (ER). Contudo a evolução do processo deve ser realizado através das outras abordagens.*

Em terceiro lugar, a análise da figura 23 permite verificar uma ligação preferencial das estratégias de desenvolvimento de software a determinados paradigmas de programação. Esta ligação caracteriza-se naturalmente pela existência de transformações homogéneas. Observa-se então que a abordagem formal prevê a implementação em linguagens funcionais, lógicas ou imperativas, a estratégia não formal adapta-se às linguagens imperativas ou relacionais e a abordagem orientada aos objectos prevê a implementação numa linguagem orientada aos objectos. Conclui-se que, *um dos factores de ponderação na selecção da melhor abordagem para a especificação dos requisitos (ER) e a especificação do software (ES), deve ser a linguagem de programação (S) a usar*

Finalmente, relacionando a figura 23 com a classificação (quanto à abordagem e áreas de aplicação) das linguagens de programação apresentada (ver secção 3.4.4.), podem tirar-se algumas conclusões sobre que estratégia de desenvolvimento de software adoptar, perante um problema em determinada área de aplicação. Por exemplo, para um problema na área de inteligência artificial, adaptam-se linguagens de programação funcionais ou lógicas. Estes paradigmas de programação preveem a utilização de uma abordagem formal para o desenvolvimento de software. Considera-se então que, *a área de aplicação do software está intimamente ligada à selecção da linguagem de programação (S), que como referido deve influenciar o tipo de abordagem utilizada para o desenvolvimento de software*

Um facto curioso que sobressai da revisão realizada é a *existência de um elevado número de técnicas de representação, não se verificando o mesmo quanto às transformações*. Notou-se também que várias das transformações analisadas possuem apenas um carácter informal

Afigura-se interessante uma passagem futura desta perspectiva teórica para uma visão prática, ou seja, definir e preferencialmente automatizar novas transformações, permitindo uma adaptação mais correcta das representações à área de aplicação do software. O objectivo

será flexibilizar o processo de desenvolvimento de software em relação às abordagens utilizadas e tipos de linguagens de programação. Verifica-se a necessidade de uma melhor sistematização das transformações existentes, bem como a definição de novas transformações

Por outro lado, teria interesse analisar o modo como o desenvolvimento de software é realizado nas organizações, em relação ao modelo de representações / transformações aqui analisado, isto é, saber que representações e transformações são efectivamente utilizadas na prática. O objectivo seria verificar o modo como esta problemática é tratada nas organizações e validar as conclusões apresentadas.

(esta página foi deixada propositadamente em branco)

Bibliografia

Abrial, J. R., Schuman, S. A., e Meyer, B., *Specification Language.*, R. McKeag and A. Macnaghten, Cambridge University Press, 1980.

Agresti, W., *New Paradigms for Software Development*, IEEE Comp. Society Press, 1986.

Alavi, M., "An Assessment of the Prototyping Approach to Information Systems Development" *Communications of the ACM*, Vol. 27, Nº 6, (1984), pg.556-563.

Ambler, A., Burnett, M., e Zimmerman, B., "Operational versus Definitional: A Perspective on Programming Paradigms" *IEEE Computer*, Vol. Setembro, (1992), pg.28-43.

ANSI/IEEE Std. 830., *IEEE Guide for Software Requirements Specifications*, The Institute of Electrical and Electronics Engineers, Inc., 1984.

ANSI/IEEE Std. 729, *IEEE Standard Glossary of Software Engineering Terminology*, The Institute of Electrical and Electronics Engineers, Inc., 1983.

Bailin, S., "An Object-Oriented Requirements Specification Method" *Communications of the ACM*, Vol. 32, Nº 5, (1989), pg.608-623.

Balzer, R., "Transformational Implementation: An Example" *IEEE Transactions on Soft. Eng.*, Vol. SE-7, Nº 1, (1981), pg.3-14.

Balzer, R., e Goldman, N., *Principles of a Good Specification and their Implications for Specification Languages*, Addison-Wesley, 1986.

Balzer, R., Goldman, N., e Wile, D., "On the Transformational Implementation Approach to Programming" *Proc. of 2th International Conf. on Software Eng.*, (1976), pg.337-344.

Basili, V., e Musa, J., "The Future Engineering of Software: A Management Perspective" *IEEE Computer*, Vol. Setembro, (1991), pg.90-96.

Bauer, F., Moller, B., Partsch, H., e Pepper, P., "Formal Program Construction by Trans. - Computer-Aided, Intuition-Guided Programming" *IEEE Transactions on Software Engineering*, Vol. 15, Nº 2, (1989), pg.165-179.

Bauer, F., "Programming as an Evolutionary Process" *Lecture Notes in Computer Science*, Vol. 46, (1976), pg.153-182.

Bell, T., e Bixler, D., "A Flow Oriented Requirement Statement Language." *Proc. of the Symposium on Computer Soft. Engineering, Polytechnic Press*, (1976), pg.109-122.

Bidoit, M., Gaudel, M., e Mauboussin, A., "How to Make Algebraic Specifications more Understandable?" *Lecturer Notes in Computer Science*, Vol. 394, (1989), pg.31-68.

Birrel, N., e Ould, M., *A Pratical Handbook for Software Development*, Cambridge University Press, 1985.

Bobrow, D., "If PROLOG is the Answer, What is the Question?" *Proceedings of Fourth Generation Computer Systems*, (1984), pg.138-148.

Boehm, B., "A Spiral Model of Software Development and Enhancement" *Software Engineering Project Management*, (1987), pg.128-142.

Boehm, B., Gray, T., e Seewaldt, T., "Prototyping Versus Specifying: A Multiproject Experiment" *IEEE Trans. on Software Eng.*, Vol. SE-10, Nº 3, (1984), pg.290-302.

Boehm, B., "Software Engineering" *IEEE Transactions on Computers*, Vol. C-25, Nº 12, (1976), pg.1226-1241.

Bollinger, T., e McGowan, C., "A Critical Look as Software Capability Evaluations" *IEEE Software*, Vol. Julho, (1991), pg.25-41.

Booch, G., "Object-Oriented Development" *IEEE Transactions on Software Engineering*, Vol. SE-12, Nº 2, (1986), pg.211-221.

Booch, G., *Object-Oriented Design with Applications*, Benjamin/Cummings, 1991.

Brackett, J., *Software Requirements*, SEI Curriculum Module SEI-CM-19-1.2, 1990.

Britton, K., e Parnas, D., *A-7E Software Module Guide*, Naval Research Lab., Rep. 4702, 1981.

Brooks, F., "No Silver Bullet, Essence and Accidents of Software Engineering" *IEEE Computer*, Vol. Abril, (1987), pg.10-19.

BS6719, *British Standard Guide to Specifying User Requirements for a Computer-Based System*, British Standards Institute, 1986.

Burgess, R., *Structured Program Design using JSP.*, Hutchinson, 1987.

Caine, S., e Gordon, E., "PDL - A Tool for Software Design." *Proceedings of the AFIPS National Computer Conference. AFIPS Press.*, Vol. 44, (1975), pg.271-276.

Cameron, J., "An Overview of JSD" *IEEE Transactions on Software Engineering*, Vol. SE-12, Nº 2, (1986), pg.222-240.

Cameron, J., Campbell, A., e Ward, P., "Comparing Software Development Methods: Example" *Information and Software Technology*, Vol. 33, Nº 6, (1991), pg.386-402.

Capretz, L., e Lee, P., "Object-Oriented Design: Guidelines and Techniques" *Information and Software Technology*, Vol. 35, Nº 4, (1993), pg.195-206.

Champeaux, D., e Faure, P., "A Comparative Study of Object-Oriented Analysis Methods." *J. Object-Oriented Programming*, Vol. 5, Nº 1, (1992), pg.21-33.

Charette, R., *Software Engineering Environments*, McGraw Hill, 1986.

Chen, M., e Norman, R., "A Framework for Integrated CASE" *IEEE Software*, Vol. Março, (1992), pg.18-22.

Chen, P., "Entity-Relationship to Data Modeling" *System and Software Requirements Engineering. IEEE Computer Society Press Tutorial*, (1990), pg.239-243.

Chen, P., "The Entity Relationship Model - Toward a Unified View of Data" *ACM Transaction on Data Base Systems*, Vol. 1, Nº 1, (1976), pg.6-36.

Coad, P., e Yourdon, E., *Object-Oriented Design*, Prentice-Hall, 1991.

Coad, P., "Object-Oriented Patterns" *Commun. ACM*, Vol. 35, Nº 9, (1992), pg.152-159.

Codd, E., "A Relational Model of Data for large shared Data Banks." *Communiactions of the ACM*, Vol. 13, (1970), pg.377-387.

Cox, B., *Programação Orientada para Objecto*, Addison Wesley, 1991.

Cox, B., "Planning the Software Industrial Revolution" *IEEE Software*, Vol. Novembro, (1990), pg.25-33.

Crosby, P., *Quality Is Free*, McGraw-Hill, 1979.

Curtis, B., Kellner, M., e Over, J., "Process Modeling" *Commun. of the ACM*, Vol. 35, Nº 9, (1992), pg.75-90.

Davies, C., e Layzell, P., *The Jackson Approach to System Development*, Chartwell Bratt, 1993.

Davis, A., "A Comparison of Techniques for the Specification of External System Behavior" *Communications of the ACM*, Vol. 31, Nº 9, (1988), pg.1098-1115.

Davis, A., *Software Requirements*, Prentice-Hall International, 1990.

Davis, A., Bersoff, E., e Comer, E., "A Strategy for Comparing Alternative Software Development Life Cycle Models" *IEEE Trans. on Software Eng.*, Vol. 14, Nº 10, (1988), pg.1453-1460.

DeMarco, T., *Structured Analisys and System Specification*, Prentice-Hall, 1978.

Deming, W., *Quality, Productivity, and Competitive Position*, Cambridge, MA: Massachusets Institute of Technology Center for Advanced Engineering Study., 1982.

Dijkstra, E., "Structured Programming" *Software Eng., Concepts and Technics*, (1976).

DOD-STD-2167A, *Military Standard: Defense System Software Development*, U.S. Department of Defense, 1988.

DOD-STD-7935A, *Automated Information Systems (AIS) Documentation Standards*, U.S. Department of Defense, 1988.

Dorfman, M., "System and Software Requirements Engineering" *Tutorial: System and Software Requirements Engineering. IEEE Computer Society Press*, (1990), pg.4-16.

Downs, E., Clare, P., e Coe, I., *Structured Systems Analysis and Design Method*, 2ª ed., Prentice-Hall, 1992.

Elmasri, R., e Navathe, S., *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, 1989.

ESA PSS-05-0, *The Software Life Cycle: The User's Requirements Definition Phase: and the Software Requirements Definition Phase*, European Space Agency, 1987.

Falkenberg, E., "Foundation of the Conceptual Schema Approach of Information Systems Management" *Lecture Notes for Database Management and Applications*, (1982).

Fichman, R., e Kemerer, C., "Object-Oriented and Conventional Analysis and Design Methodologies" *IEEE Computer*, Vol. Outubro, (1992), pg.22-39.

FIPS-Pub-101, *Guideline for Lifecycle Validation, Verification and Testing of Computer Software*, National Bureau of Standards, 1983.

Forte, G., e Norman, R., "A Self-Assessment by the Software Engineering Community" *Communications of the ACM*, Vol. 35, Nº 4, (1992), pg.28-32.

France, R., "Semantically Extended Data Flow Diagrams: A Formal Specification Tool" *IEEE Transactions on Software Engineering*, Vol. 18, Nº 4, (1992), pg.329-346.

Fraser, M., Kumar, K., e Vaishnavi, V., "Informal and Formal Requirements Specification Languages: Bridging the Gap" *IEEE Transactions on Software Engineering.*, Vol. 17, Nº 5, (1991), pg.454-465.

Freeman, P., "Fundamentals of Design" *IEEE Tutorial on Software Design Techniques. IEEE Computer Society Press*, (1983), pg.2-22.

Friedman, L., *Comparative Programming Languages - Generalizing the Programming Function.*, Prentice-Hall, 1991.

Gane, C., e Sarson, T., *Analise Estruturada de Sistemas*, 13ª ed., Livros Tecnicos e Cientificos Editora Ltda, 1983.

Gaudel, M., "Towards Structured Algebraic Specification" *Esprit 85 Status Report, North-Holland*, (1985), pg.493-510.

Ghezzi, C., e Jazayeri, M., *Programming Language Concepts*., John Wiley, 1987.

Ghezzi, C., Jazayeri, M., e Mandrioli, D., *Fundamentals of Software Engineering*, Prentice-Hall, 1991.

Gomaa, H., "The Impact of Prototyping on Software System Engineering" *Tutorial: System and Software Requirements Eng. IEEE Computer Society Press*, (1990), pg.543-552.

Guttag, J., "Abstract Data Types and the Development of Data Structures" *Communications of the ACM*, Vol. 32, Nº 10, (1977), pg.1164-1173.

Guttag, J., e Horning, J., "Report on the Larch Shared Language" *Science of Computer Programming*, Vol. 6, (1986), pg.103-134.

Hailpern, B., "Multiparadigm Languages" *IEEE Software*, Vol. Janeiro, (1986), pg.6-9.

Hall, A., "Seven Myths of Formal Methods" *1990*, Vol. Setembro, (1990), pg.11-19.

Hayes, I., *Specification Case Studies*., Prentice-Hall International, 1987.

Henderson-Sellers, B., e Edwards, J., "The Object Oriented Systems Life Cycle" *Communications of the ACM*, Vol. 33, Nº 9, (1990), pg.143-159.

Henderson-Sellers, B., e Constantine, L., "Object-Oriented Development and Functional Decomposition." *J. Object-Oriente Prog.*, Vol. 33, Nº 9, (1991), pg.142-169.

Hester, S., Parnas, D., e Utter, D., "Using Documentation as a Software Design Medium" *The Bell System Technical Design*, (1981), pg.60.

Hodgson, R., e Dunne, S., "Process, Data and Behavior (Perspectives on System Development)" *Information and Soft. Technology*, Vol. 32, Nº 8, (1990), pg.539-544.

Hull, M., e O'Donoghue, P., "Family Relationship between Requirements and Design Specification Methods" *The Computer Journal*, Vol. 36, Nº 2, (1993), pg.153-167.

Humphrey, W., *Managing the Software Process*, Addison-Wesley, 1989.

Humphrey, W., "Characterizing the Software Process: A Maturity Framework" *IEEE Software*, Vol. Março, (1988), pg.73-79.

Humphrey, W., e Curtis, B., "Comments on 'A Critical Look'" *IEEE Software*, Vol. Julho. (1991), pg.42-46.

Humphrey, W., Snyder, T., e Willis, R., "Software Process Improvement at Hughes Aircraft" *IEEE Software*, Vol. Julho, (1991), pg.11-23.

Ince, D., "Formal Methods: Set Piece." *Datalink*, Vol. 1, (1989).

Ingervaldsson, L., *JSP: A Pratical Method of Program Design.*, Chartwell Bratt, 1979.

Jackson, M., *System Development*, Prentice-Hall, 1983.

Jacobson, I., Christerson, M., Jonsson, P., e Overgaard, G., *Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley, 1992.

Jones, C., *Systematic Software Development using VDM*, 2ª ed., Prentice-Hall, 1990.

Jones, C., e Shaw, R., *Case Studies in Systematic Software Development*, 1ª ed., Prentice-Hall, 1990.

Karam, G., e Casselman, R., "A Cataloging Framework for Software Development Methods" *IEEE Computer*, Vol. Fevereiro, (1993), pg.34-47.

Kellner, M., e Humphrey, W., "Software Process Modeling: Principles of Entity Process Models" *Proc. 11th Intern. Conf. on Software Eng., IEEE Computer Society Press*, (1989).

Kemmerer, R., "Integrating Formal Methods into the Development Process" *IEEE Software*, Vol. Setembro, (1990), pg.37-50.

Kimura, T., Choi, J., e Mack, J. "Show and Tell: A Visual Programming Language" *Visual Programming Environments*, (1990), pg. 397-404.

Kowalski, R., "The Origins of Logic Programming" *Byte*, Vol. 8, (1985), pg.192-197.

Loy, P., "A Comparision of Object-Oriented and Structured Development Methodologies." *ACM SIGSoft Software Eng. Notes*, Vol. 15, Nº 1, (1990), pg.44-48.

Martin, J., *Information Engineering*, Prentice-Hall, 1990.

Martin, J., e McClure, C., *Técnicas Estruturadas e CASE*, Makron Books, 1991.

Martin, J., *Computer Data Base Organization*, Prentice-Hall, 1982.

Maulsby, D., e Witten, I., "Inducing Programs in a Direct-Manipulation Environment" *Proc. CHI'89*, (1989), pg. 57-62.

McDermid, J., *Software Engineering Reference Book*, Butterworth-Heinemann Ltd, 1991.

McIlroy, M., "Mass Produced Software Components" *Software Engineering (NATO Science Committe)*, (1969), pg.138-150.

Meyer, B., "On Formalism in Specification" *IEEE Software*, Vol. Janeiro, (1985), pg.6-26.

Meyer, B., "Tools for the New Culture: Lessons from the Design of the Eiffel Libraries" *Communications of the ACM*, Vol. 35, Nº 9, (1990), pg.68-88.

Monarchi, D., e Puhr, G., "A Reasearch Topology for Object-Oriented Analysis and Design" *Communications of the ACM*, Vol. 35, Nº 9, (1992), pg.35-47.

Morgan, C., e Sufrin, B., "A Formal Specification of the Unix File System," *IEEE Transactions on Software Engineering*, Vol. SE-10, (1984).

Moynihan, T., "Modelling the Software Process in Terms of the System Representations and Transformations Steps Used" *Information and Software Technology.*, Vol. 35, Nº 3, (1993), pg.181-188.

Myers, G., *Reliable Software Through Composite Design*, Petrocelli/Charter, 1975.

Nerson, J., "Applying Object-Oriented Analysis and Design" *Communications of the ACM*, Vol. 35, Nº 9, (1992), pg.63-74.

Nierstrasz, O., Gibbs, S., e Tsichritzis, D., "Component-Oriented Software Development" *Communications of the ACM*, Vol. 35, Nº 9, (1992), pg.160-165.

Nijssen, G., e Halpin, T., *Conceptual Schema and Relational Database Design*, Prentice-Hall, 1989.

Nijssen, G., "On Experience with Large Scale Teaching and Use of Fact-Based Conceptual Schemas in Industry and University" *Database Semantics: Proc. IFIP Conf. on Database Semantics*, (1985).

NIST/ECMA, *Reference Model for Frameworks of Software Engineering Environments*, National Institute of Standards and Technology, 1991.

Norman, R., e Chen, M., "CASE Environments" *IEEE Software*, Vol. Março, (1992), pg.13-16.

Norris, M., "Z (A Formal Specification Method)" *A Debrief Report, STARTS, The National Computing Centre Limited*, (1986).

Oliveira, J., *Desenvolvimento Formal de 'Software'*, Universidade do Minho, 1992.

Olle, W., e al., *Information Systems Methodologies*, Addison-Wesley, 1988.

Page-Jones, M., *The Pratical Guide to Structured System Design*, 2ª ed., Prentice-Hall, 1988.

Partsch, H., e Steinbruggen, R., "Program Transformations Systems" *Computing Surveys*, Vol. Setembro, (1983), pg. 199-236.

Pfleeger, S., *Softawre Engineering, the Production of Quality Software*, 2ª ed., Maxwell MacMillan, 1991.

Placer, J., "The Multiparadigm Language G" *Computer Languages*, Vol. 16, Nº 3/4, (1991), pg.235-258.

Pokkunuri, B., "Object Oriented Programming" *ACM SIGPLAN Notices*, Vol. 24, Nº 11, (1989), pg.96-101.

Prehn, S., "From VDM to RAISE" *Lecturer Notes in Computer Science, VDM 1987*, Vol. 252, (1987), pg.141-149.

Pressman, R., *Software Engineering, a Practitioner's Approach*, 3ª ed., Mcgraw-Hill, 1992.

Rajlich, V., "Paradigms for Design and Implementation in Ada" *Communications of the ACM*, Vol. 28, Nº 7, (1985), pg.718-727.

Rombach, H., *Software Spec: A Framework*, SEI Curriculum Module SEI-CM-11-1.2, 1990.

Royce, W., "Managing the Development of Large Software Systems: Concepts and Techniques" *Proceedings of WESCON*, (1970).

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., e Lorensen, W., *Object-Oriented Modeling and Design*, 1ª ed., Prentice Hall International Editions, 1991.

Sage, A., e Palmer, J., *Software Systems Engineering*, Wiley, 1990.

Semadas, A., "Objectos, Conceitos e Construções" *Proceedings of OOP'93*, (1993).

Shaw, M., "Prospects for an Engineering Discipline of Software" *IEEE Software*, Vol. Novembro, (1990), pg.15-24.

Shemer, I., "Systems Analysis: A Systemic Analysis of a Conceptual Model" *Communications of the ACM*, Vol. 30, Nº 6, (1987), pg.506-512.

Shlaer, S., e Mellor, S., *Object Oriented Analysis: Modeling the World in Data*, Yourdon Press, 1988.

Sinderen, M., Pires, L., e Vissers, C., "Protocol Design and Implementation Using Formal Methods" *The Computer Journal*, Vol. 35, Nº 5, (1992), pg.478-491.

Skidmore, S., Farmer, R., e Mills, G., *SSADM Models and Methods*, NCC Blackwell, 1992.

Sommerville, I., *Software Engineering*, 4ª ed., Addison-Wesley, 1992.

Spivey, J. M., *The Z Notation, A Reference Manual.*, Prentice-Hall, 1989.

STARTS-Guide, *Requirements Definition and Design*, 2ª ed., National Computer Center, Ltd., 1987.

Stevens, W., Myers, G., e Constantine, L., "Structured Design" *IBM Systems Journal*, Vol. 13, Nº 2, (1974), pg.115-139.

Stroustrup, B., "What is Object-Oriented Programming?" *IEEE Software*, Vol. Maio, (1988), pg.10-19.

Sufrin, B., *Formal System Specification: Notation and Example.*, Cambridge University Press, 1983.

Sutcliffe, A., *Jackson System Development*, Prentice-Hall, 1988.

Sutcliffe, A., "Object-Oriented Systems Development: Survey of Structured Methods" *Information and Software Technology*, Vol. 33, Nº 6, (1991), pg.433-442.

SWAP-DIP-P100, *Product Specification Document Standards*, U.S. Government, 1989.

Swartout, W., e Balzer, R., "On the Inevitable Interwining of Specification and Implementation" *Communications of the ACM*, Vol. 25, Nº 7, (1982), pg.438-440.

Taylor, E., *An Interim Report on Engineering Design*, Massachusetts Institute of Technology, 1959.

Thayer, R., e Dorfman, M., *System and Software Requirements Engineering*, IEEE Computer Society Press Tutorial, 1990.

Thomas, D., "What's in an Object?" *Byte*, Vol. 3, (1989), pg.231-240.

Tsai, J., e Ridge, J., "Intelligent Support for Specifications Transformation" *IEEE Software*, Vol. Novembro, (1988), pg.28-35.

Ullman, J., *Principles of Database Systems*, Computer Science Press, 1982.

Vessey, I., Jarvenpaa, S., e Tractinsky, N., "Evaluation of Vendor Products: CASE Tools as Meth. Companions" *Communications of the ACM*, Vol. 35, Nº 4, (1992), pg.91-103.

Viehstaedt, G., e Ambler, A., "Visual Representation and Manipulation of Matrices" *J. Visual Language and Computing*, Vol. 3, Nº 3, (1992).

Ward, P., "How to Integrate Object Orientation with Structured Analysis and Design" *IEEE Software*, Vol. Março, (1989), pg.74-82.