

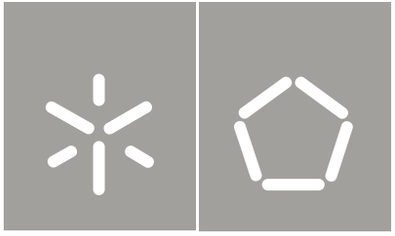
Universidade do Minho
Escola de Engenharia

Nuno Filipe Gomes Cardoso

Middleware e Ferramentas para desenvolvimento de Sistemas de Vigilância para Segurança, Controlo e Conforto (SVSC^{2}- ^{M}Toolkit)

Janeiro de 2013

Middleware e Ferramentas para desenvolvimento de Sistemas de Vigilância para Segurança, Controlo e Conforto (SVSC^{2}- ^{M}Toolkit)
Nuno Filipe Gomes Cardoso
UMinho | 2013



Universidade do Minho
Escola de Engenharia

Nuno Filipe Gomes Cardoso

Middleware e Ferramentas para desenvolvimento de Sistemas de Vigilância para Segurança, Controlo e Conforto (SVSC^{2}- ^{M}Toolkit)

Tese de Doutoramento
Informática Industrial

Trabalho efectuado sob a orientação do
Professor Doutor José Araújo Mendes
Professor Doutor Adriano José Conceição Tavares

Janeiro de 2013

DECLARAÇÃO

Nuno Filipe Gomes Cardoso

Endereço electrónico: id2247@alunos.uminho.pt Telefone: 936 242 711

Número do Bilhete de Identidade: 11019877

Título tese:

Middleware e Ferramentas para desenvolvimento de **S**istemas de **V**igilância para **S**egurança, **C**ontrolo e **C**onforto (**SVSC**^{2} - ^{M}**Toolkit**).

Orientadores:

Professor Doutor José Araújo Mendes

Professor Doutor Adriano José Conceição Tavares

Ano de conclusão: 2013

Designação do Doutoramento:

Informática Industrial

Nos exemplares das teses de doutoramento ou de mestrado ou de outros trabalhos entregues para prestação de provas públicas nas universidades ou outros estabelecimentos de ensino, e dos quais é obrigatoriamente enviado um exemplar para depósito legal na Biblioteca Nacional e, pelo menos outro para a biblioteca da universidade respectiva, deve constar uma das seguintes declarações:

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE/TRABALHO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

Universidade do Minho, ___/___/_____

Assinatura: _____

Agradecimentos

As primeiras palavras de agradecimento vão para os meus orientadores científicos, o Doutor José Mendes e o Doutor Adriano Tavares, pela orientação e apoio dado ao longo da execução deste projeto.

Agradeço também de uma forma muito especial ao Doutor Jorge Cabral, que muito contribuiu para que este projeto se tornasse uma realidade.

Agradeço a todos os alunos e amigos do ESRG (*Embedded Systems Research Group*), em especial ao Filipe Salgado, ao Paulo Garcia e ao Tiago Gomes.

Agradeço também à Fundação para a Ciência e Tecnologia, por ter concedido e financiado a bolsa de Doutoramento e à Universidade do Minho, em especial ao Departamento de Eletrónica e Computadores.

Gostaria também de deixar uma nota de agradecimento muito especial aos meus pais, Bernardino e Maria de Lurdes. Ao meu pai por me ensinar através da natureza, as coisas simples e essenciais da vida. À minha mãe pelo esforço, carinho e dedicação prestados ao longo da minha vida.

Os meus votos de agradecimento são extensíveis também à minha irmã Arminda, que sempre me acompanhou e motivou ao longo desta caminhada. Como foram importantes as palavras que um dia me disseste: "só consegues mudar algo, se estiveres presente".

Por fim, agradeço de uma forma muito especial à minha namorada Andrea, por estar sempre presente nos bons e maus momentos. O quanto foste compreensiva e motivadora! Agradeço também o facto de um dia me teres oferecido um livro, porque tinhas a noção de que, para escrever bem, é preciso ler!

A todos, amigos e familiares, muito obrigado!

Guimarães, 31 de Outubro de 2012.

Resumo

Devido a vários fatores socioeconómicos, existe atualmente uma enorme necessidade de se utilizarem sistemas de videovigilância inteligentes. O desenvolvimento deste tipo de sistemas tem-se tornado numa tarefa complexa, devido à necessidade de desenvolver novos produtos com um elevado número de diferentes requisitos.

A complexidade está relacionada com o elevado nível de variabilidade/heterogeneidade que necessita de ser gerida. A complexidade deve-se à gestão da variabilidade em tarefas relacionada com: a captura de imagens, processamento, compressão, comunicações e visão por computador. Mais, os sistemas atuais devem correr em diferentes plataformas de *hardware*, que vão desde os PCs *desktop* até sistemas embebidos, telefones móveis, etc.

Para resolver o problema da complexidade e a enorme necessidade de sistemas de videovigilância adaptados à aplicação, com um reduzido *time-to-market*, esta tese propõe explorar o domínio da videovigilância como uma família de produtos, que podem ser desenvolvidos seguindo uma filosofia *pay-as-you-go* ao invés de desenvolver produtos *ex-nihilo*.

De forma a satisfazer os requisitos apresentados anteriormente, é proposta uma *framework* generativa de videovigilância. Na fase de modelação da *framework* generativa, é usada uma combinação de tecnologias e técnicas tais como, *Software Product Line* (SPL), *Model Driven Development* (MDD) e Agile, de modo a gerir a variabilidade existente ao nível das funcionalidades do sistema. Na fase da implementação das funcionalidades, são utilizados componentes totalmente configuráveis, desenvolvidos em C++ *template metaprogramming*, de modo a gerir a variabilidade do código do sistema e a reutilização dos componentes.

A *framework* de videovigilância proposta apresenta vantagens comparativamente com os métodos tradicionais, principalmente, reduz o *time-to-market*, causa baixo impacto no desempenho e permite gerir melhor a variabilidade.

Abstract

Due to a variety of socio-economic factors, currently there is a growing demand for intelligent video surveillance systems. The development of such systems is becoming increasingly complex since there is a high demand for rapid deployment of novel products with a rising number of different requirements.

The complexity arises from the high level of variability/heterogeneity that needs to be managed. The increasing complexity is induced by the variability in tasks related to image capturing, processing, compression, communications and computer vision. Furthermore, it is now expected that systems should run in different hardware platforms, ranging from desktop PCs to low cost embedded boards, mobile phones, etc.

To tackle the growing complexity and huge demand for application-tailored video surveillance systems, along with very tight time-to-market expectations, this thesis proposes exploiting the video surveillance domain as families of systems that can be developed following a pay-as-you-go fashion rather than developing products ex-nihilo.

To better accommodate all the above listed requirements, a new video surveillance fully-reconfigurable generative framework is proposed. In the framework's modeling stage, the combination and interoperability of Software Product Line (SPL) technologies, Model-Driven Development (MDD) and Agile Development were promoted, in order to manage the existing variability at the system functionality level. At the framework's implementation stage, fully-customizable components, developed in C++ template metaprogramming, were used, in order to provide code variability management mechanisms and to promote high component re-use.

The proposed video surveillance generative framework presents several advantages when compared to the traditional methods, namely, short time-to-market, low performance overhead, and variability management.

Conteúdo

Agradecimentos **iii**

Resumo **v**

Abstract **vii**

Acrónimos **xix**

1 Introdução **1**

1.1 Motivações 5

1.2 Objetivos e Contribuições 6

1.3 Organização da Tese 6

2 Estado da Arte **9**

2.1 Introdução 9

2.2 Evolução dos Sistemas de Videovigilância 10

2.2.1 Sistemas de Videovigilância de 1ª Geração 10

2.2.2 Sistemas de Videovigilância de 2ª Geração 11

2.2.3 Sistemas de Videovigilância de 3ª Geração 12

2.3 Arquitetura Atual dos Sistemas de Videovigilância 13

2.3.1 Principal Problema dos Atuais Sistemas de Videovigilância . 15

2.4 *Software Product Line* 17

2.4.1 Conceitos e Terminologia 18

2.4.2 Especificação do Espaço do Problema 19

2.4.3 Implementação do Espaço da Solução 21

2.5 Trabalhos Relacionados 35

2.6 Conclusão 37

3 C++ em Ambientes Embebidos **39**

3.1 Introdução 39

3.2 Linguagem de Programação C++ 40
3.2.1 Principais Conceitos da Linguagem 40
3.3 C++ em Sistemas Embebidos 43
3.3.1 Classes 44
3.3.2 *Namespace* 47
3.3.3 Funções *Inline* 49
3.3.4 *Overloading* de Operadores 51
3.3.5 Construtor e Destrutor 52
3.3.6 Referências 54
3.3.7 Herança Única 56
3.3.8 Herança Múltipla 58
3.3.9 Funções Virtuais 63
3.3.10 *Templates* 68
3.3.11 Exceções 71
3.3.12 RTTI 74
3.4 Conclusão 78

4 *DirectShow* em Sistemas Embebidos 81
4.1 Introdução 81
4.2 Conceitos Básicos do *DirectShow* 82
4.2.1 Arquitetura Modular 83
4.2.2 Principais Componentes do *DirectShow* 83
4.3 Implementação do *DirectShow* 87
4.3.1 Interfaces e Múltipla Herança 87
4.3.2 Criação Dinâmica dos Objetos 95
4.3.3 Tratamento de Exceções 98
4.4 Impacto do *DirectShow* em Sistemas Embebidos 99
4.4.1 Filtro de transformação *CMjpegDec* 100
4.5 Conclusão 102

5 *Frameworks* TMP e a Gestão da Variabilidade 103
5.1 Introdução 103
5.2 *Template Metaprogramming* 104
5.2.1 Blocos Básicos do TMP 104
5.3 *Template Metaprogramming Frameworks* 108
5.3.1 *Boost Metaprogramming Library* 109
5.3.2 *Generic Image Library* 118
5.4 Conclusão 124

6 *Framework* Generativa de Videovigilância **127**
6.1 Introdução . . . 127
6.2 *Software Product Line* . . . 128
6.2.1 Domínio de Engenharia . . . 128
6.2.2 Engenharia da Aplicação . . . 177
6.3 *Model-Driven Template Metaprogramming* . . . 185
6.3.1 *Template Markup Language* (TMPxML) . . . 186
6.4 Conclusão . . . 190

7 Resultados Experimentais **191**
7.1 Introdução . . . 191
7.2 Ambiente de Testes . . . 192
7.2.1 Ferramentas Utilizadas . . . 193
7.3 Métricas de Teste . . . 193
7.4 Testes Realizados . . . 194
7.5 Filtro de Transformação *CJPEGDecTransform* . . . 195
7.5.1 Extração das Funcionalidades . . . 196
7.5.2 Testes das Características Não Funcionais . . . 198
7.5.3 Testes de Gestão do Código . . . 203
7.6 Conclusão . . . 204

8 Conclusão e Trabalho Futuro **207**
8.1 Conclusão . . . 207
8.2 Trabalho Futuro . . . 211

Bibliografia **212**

A Implementação de um Filtro **223**
A.1 Diagrama de Classes . . . 223
A.2 Métodos de Captura . . . 225
A.3 Formatos de Captura . . . 226
A.4 Estrutura *Template CV4L2SourceImpl* . . . 226
A.5 Implementação do Pino de Saída . . . 228

B *Schema* XML da linguagem TMPxML **231**
B.1 Elemento Module . . . 232
B.2 Elemento Template . . . 233
B.3 Elemento Struct e o Elemento Class . . . 234

B.4 Elemento Parameter 235
B.5 Elemento Extension 235
B.6 Elemento Code 236
B.7 Elemento Specialization 237
B.8 Elemento Typedef 238
B.9 Elemento Variable 239
B.10 Função Factorial 239

Lista de Figuras

1.1 Variabilidade existente ao nível das funcionalidades e tecnologias utilizadas nos sistemas de videovigilância de terceira geração. 2

2.1 Exemplo de um sistema de videovigilância de 1ª geração. Imagem extraída de (Axis). 11
2.2 Exemplo de um sistema de videovigilância de 2ª geração. Imagem extraída de (Axis). 12
2.3 Exemplo de um sistema de videovigilância de 3ª geração. Imagem extraída de (Axis). 13
2.4 Arquitetura do sistema de videovigilância PRISMATICA. Imagem adaptada de (Lo et al., 2003) 14
2.5 Variabilidade existente ao nível das funcionalidades e tecnologias utilizadas nos vários nós inteligentes que constituem os sistemas de videovigilância de 3ª geração. 16
2.6 Relação que existe entre a gestão da variabilidade e o desempenho de um sistema de videovigilância de 3ª geração. 16
2.7 Etapas do estágio do domínio da engenharia do SPL. Imagem adaptada de (Lohmann, 2009). 18
2.8 Etapas do estágio da engenharia da aplicação do SPL. Imagem adaptada de (Lohmann, 2009). 19
2.9 Tipos de funcionalidades mais utilizadas num diagrama de funcionalidades. Imagens adaptadas de (Lohmann, 2009). 20
2.10 Exemplo típico de um diagrama de funcionalidades. Diagrama extraído de (Czarnecki and Eisenecker, 1999) 21

3.1 Disposição dos dados da classe Point2D em memória. 45
3.2 Disposição dos dados da classe Point3D em memória. 57
3.3 Disposição dos dados da classe SharedPoint3D em memória. 59
3.4 Disposição dos dados da classe Point3D em memória. 64

4.1 Relação entre uma aplicação *DirectShow* e os vários componentes do *DirectShow*. Imagem adaptada de Pesce (2002). 82
4.2 Exemplo de uma configuração do *filter graph*. 84
4.3 Organização dos dados da classe CFilterGraph em memória. 90
4.4 Diagrama de classes da classe CBaseFilter. 91
4.5 Diagrama de classes das classes que servem de base de implementação dos filtros. 92
4.6 Diagrama de classes da classe CBasePin. 94
4.7 Diagrama de classes das classes que servem de base de implementação dos pinos. 94

5.1 Diagrama com as iterações necessárias para calcular o factorial de 3. 108
5.2 Variabilidade que existe nos formatos de representação de uma imagem. 119
5.3 Imagens de cada um dos estágios intermédios de processamento. Imagem extraída de (Bourdev). 124

6.1 Âmbito do Domínio da Videovigilância. 129
6.2 Diagrama de funcionalidades do *Filter Graph*. 130
6.3 Diagrama de funcionalidades do filtro de entrada V4L2. 132
6.4 Diagrama de funcionalidades do filtro de transformação *JPEG Encoder*. 133
6.5 *Samplings* do formato JPEG. 134
6.6 Diagrama de funcionalidades do filter de saída Ficheiro. 134
6.7 Diagrama de funcionalidades do *filter graph* no FMP. 135
6.8 Exemplo de uma configuração do *filter graph*. 137
6.9 Diagrama de classes do *template* artefacto *filter graph*. 138
6.10 Processo iterativo do método StartFilters do exemplo do *filter graph* da Figura 6.8. 143
6.11 Exemplo da Figura 6.8 com os pinos. 145
6.12 Diagrama de classes do *template* artefacto filtro. 147
6.13 Tipos de *media* de áudio/vídeo suportados por cada um dos pinos que compõem os filtros do *filter graph*. 156
6.14 Diagrama de classes do *template* artefacto pino. 157
6.15 Diagrama de classes do *template* artefacto *Intelligent Connect*. . . . 164
6.16 Processo Código *Bloat* e *Intelligent Connect*. 168
6.17 Processo iterativo de verificação da ligação entre os pinos CRTSPOutputPin e CJPEGDecInputPin. 170

6.18 Processo iterativo de verificação da ligação entre o filtro CJPEGDecTransform e o pino CXVInputPin. 172
6.19 Processo iterativo de verificação da ligação entre o filtro CRTSPSource e o filtro CXVRenderer. 174
6.20 Processo iterativo das ligações do *filter graph* da Figura 6.8. 176
6.21 Diagrama com as duas atividades principais da engenharia da aplicação. 178
6.22 Exemplo de uma configuração do *filter graph* transmitida por um *stakeholder*. 179
6.23 Seleção e configuração dos componentes no modo de modelação. . . 179
6.24 Ligação entre os componentes no modo de visualização. 180
6.25 Integração de *Agile* e SPL no processo de desenvolvimento. 181
6.26 Exemplo simples da atividade *Feature2ModelMapping*. 182
6.27 Geração automática de código TMP utilizando modelos através da linguagem TMPxML. 186
6.28 Ambiente de desenvolvimento da ferramenta de modelação TMPxML para Eclipse. 189

7.1 Ligação entre os módulos da biblioteca *libjpeg*. 197
7.2 Diagrama de funcionalidades da biblioteca *libjpeg*. 198
7.3 Tempo de execução com todas as funcionalidades selecionadas para a implementação em linguagem de programação C e C++. 199
7.4 Tempo de execução com as funcionalidades necessárias selecionadas. 200
7.5 Número de *bytes* de cada uma das funcionalidades variáveis. 201
7.6 Número de *bytes* que cada um dos módulos ocupa para cada uma das implementações da biblioteca *libjpeg*. 202
7.7 Número de *bytes* que cada um dos módulos ocupa nas versões otimizadas das implementações da biblioteca *libjpeg*. 203

A.1 Diagrama de classes do filtro de entrada V4L2. 224

B.1 *Schema* XSD do elemento *module*. 232
B.2 *Schema* XSD do elemento *template*. 233
B.3 *Schema* XSD do elemento *struct*. 234
B.4 *Schema* XSD do elemento *parameter*. 235
B.5 *Schema* XSD do elemento *extension*. 236
B.6 *Schema* XSD do elemento *code*. 236
B.7 *Schema* XSD do elemento *specialization*. 237

B.8 *Schema* XSD do elemento *typedef*. 238
B.9 *Schema* XSD do elemento *variable*. 239

Lista de Tabelas

2.1 Vantagens e desvantagens dos mecanismos de gestão da variabilidade das funcionalidades de um sistema ao nível do código. 34

5.1 *Frameworks* que utilizam *template metaprogramming*. 109

7.1 Características de *hardware* e *software* das máquinas de testes. . . . 192
7.2 Funcionalidades variáveis identificadas na biblioteca *libjpeg* 197
7.3 Módulos que implementam cada uma das funcionalidades variáveis da biblioteca *libjpeg* 197
7.4 Tamanho em *bytes* das funcionalidades comuns e variáveis da biblioteca *libjpeg*. 201
7.5 Valores de cada uma das métricas em cada um dos módulos para as diferentes implementações da biblioteca *libjpeg* 204

B.1 Elementos definidos no *Schema* XML da linguagem TMPxML. . . . 231

Acrónimos

AMD	**A**dvanced **M**icro **D**evices
AOP	**A**spect-**O**riented **P**rogramming
API	**A**pplication **P**rogramming **I**nterface
ARM	**A**dvanced **R**ISC **M**achine
AVC	**A**dvanced **V**ideo **C**oding
AVI	**A**udio **V**ideo **I**nterleave
CCTV	**C**losed-**C**ircuit **T**ele**v**ision
CODEC	**Co**der-**Dec**oder
COM	**C**omponent **O**bject **M**odel
DCT	**D**iscrete **C**osine **T**ransform
DLL	**D**ynamic-**l**ink **L**ibrary
DSP	**D**igital **S**ignal **P**rocessor
DVR	**D**igital **V**ideo **R**ecorder
EH	**E**xception **H**andling
EMF	**E**clipse **M**odeling **F**ramework
ESRG	**E**mbedded **S**ystems **R**esearch **G**roup
FMP	**F**eature **M**odeling **P**lugin
FODA	**F**eature-**O**riented **D**omain **A**nalysis
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
FT	**F**rame **T**echnology
GCC	**G**NU **C**ompiler **C**ollection
GIL	**G**eneric **I**mage **L**ibrary
HTTP	**H**yper**t**ext **T**ransfer **P**rotocol
IDA	**I**nteractive **D**is**A**ssembler
IJG	**I**ndependent **J**PEG **G**roup
IMAP	**I**nternet **M**essage **A**ccess **P**rotocol
IP	**I**nternet **P**rotocol
JPEG	**J**oint **P**hotographic **E**xperts **G**roup

LOC	**L**ines **o**f **C**ode
MDD	**M**odel **D**riven **D**evelopment
MIPS	**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
MPEG	**M**oving **P**icture **E**xperts **G**roup
MPL	**M**eta**p**rogramming **L**ibrary
MTL	**M**atrix **T**emplate **L**ibrary
NOC	**N**umber **o**f **C**lasses
NOP	**N**umber **o**f **P**ackages
OOP	**O**bject-**O**riented **P**rogramming
PC	**P**ersonal **C**omputer
POP3	**P**ost **O**ffice **P**rotocol
RGB	**R**ed **G**reen **B**lue
RTCP	**RT**P **C**ontrol **P**rotocol
RTTI	**R**un-**T**ime **T**ype **I**nformation
RTMP	**R**eal **T**ime **M**essaging **P**rotocol
RTP	**R**eal-time **T**ransport **P**rotocol
RTSP	**R**eal **T**ime **S**treaming **P**rotocol
SDK	**S**oftware **D**evelopment **K**it
SEH	**S**tructured **E**xception **H**andling
SPL	**S**oftware **P**roduct **L**ine
STL	**S**tandard **T**emplate **L**ibrary
TCP	**T**ransmission **C**ontrol **P**rotocol
TIB	**T**hread **I**nformation **B**lock
TMP	**T**emplate **M**eta**p**rogramming
TMPxML	**T**emplate **M**arkup **L**anguage
UDP	**U**ser **D**atagram **P**rotocol
VCR	**V**ideo **C**assette **R**ecorder
VSAM	**V**ideo **S**urveillance **a**nd **M**onitoring
V4L	**V**ideo for **L**inux
WAV	**W**aveform **A**udio **F**ile
XML	E**x**tensible **M**arkup **L**anguage
XSD	**X**ML **S**chema **D**efinition
XSLT	e**X**tensible **S**tylesheet **L**anguage for **T**ransformation
XVCL	**X**ML-Based **V**ariant **C**onfiguration **L**anguage
UML	**U**nified **M**odeling **L**anguage

Capítulo 1

Introdução

Na última década, os sistemas de videovigilância têm assumido cada vez mais um papel preponderante na segurança das sociedades modernas. A utilização deste tipo de sistemas apresenta como principais benefícios, nomeadamente: (i) a deteção automática de eventos anómalos, que possam pôr em risco a segurança das pessoas e (ii) a possibilidade de analisar e investigar os mesmos, após a sua ocorrência. Exemplo disso, é o impacto que os sistemas de videovigilância tiveram nas investigações dos atentados terroristas dos Estados Unidos da América a 11 de Setembro de 2001, em Madrid a 11 de Março de 2004 e em Londres a 7 de Julho de 2005 (Duque, 2009; Peixoto, 2010).

Atualmente, os sistemas de videovigilância são sistemas ubíquos, estando instalados em espaços com diferentes características e propósitos, desde espaços públicos de grandes dimensões (aeroportos, bancos, transportes públicos e espaços comerciais), até pequenos espaços privados (habitações privadas e pequeno comércio) (Ahmad et al., 2008).

Devido à necessidade da instalação deste tipo de sistemas, tanto em ambientes públicos como privados, a comunidade científica e a indústria, tem dado especial atenção à forma como os mesmos são desenhados e implementados. Como a maioria dos sistemas de videovigilância existentes são projetados de forma *ad hoc*, a sua utilização em diferentes contextos e a integração com outros sistemas, é uma tarefa complexa ou até mesmo impossível de se realizar.

Recentemente, a comunidade científica começou a considerar o uso de arquiteturas para sistemas de videovigilância inteligentes, sob a forma de *middleware* (Detmold et al., 2006a,b; Ostheimer et al., 2006) e *frameworks* (Pesce, 2002; GStreamer)

especializadas, baseadas em paradigmas multiagentes (Patricio et al., 2007; Chao and Jun, 2008). Desta forma, é possível gerir a complexidade e heterogeneidade que caracterizam este tipo de sistemas e fornecer um apoio genérico às várias arquiteturas de videovigilância existentes (Raty, 2010; Valera and Velastin, 2005).

O diagrama da Figura 1.1, apresenta, sob uma forma genérica e simplista, a variabilidade existente ao nível das funcionalidades e tecnologias utilizadas nos sistemas de videovigilância de terceira geração (Raty, 2010; Valera and Velastin, 2005):



Figura 1.1: Variabilidade existente ao nível das funcionalidades e tecnologias utilizadas nos sistemas de videovigilância de terceira geração.

Tal como ilustra o diagrama da Figura 1.1, os sistemas de videovigilância de terceira geração apresentam variabilidade ao nível: das funcionalidades de entrada (Sedky et al., 2005; Desurmont et al., 2004; Ziliani, 2005), das funcionalidades de *codecs* (Ziliani, 2005; Wijnhoven et al., 2004), das funcionalidades de gravação, das funcionalidades de processamento, das funcionalidades da arquitetura (Wijnhoven et al., 2004; Hampapur et al., 2003), das funcionalidades dos sistemas operativos e das funcionalidades do servidor.

As funcionalidades de entrada apresentam variabilidade ao nível do tipo de dispositivo e do modelo do dispositivo. No que diz respeito ao tipo de dispositivo, podem

ser classificados em: câmaras analógicas, câmaras IP, microfone, servidor, etc. Isto é, o sistema de videovigilância pode processar imagens descomprimidas provenientes de câmaras analógicas, imagens/som no formato comprimido provenientes de câmaras IP, som proveniente de um microfone, ou até mesmo, imagens/som no formato comprimido provenientes de um servidor remoto. Quanto ao modelo do dispositivo, dentro de cada tipo, existem várias marcas e modelo. A título de exemplo, uma câmara IP pode ser do fabricante *Sony* ou do fabricante *Axis*. E dentro do mesmo fabricante, também existe variabilidade ao nível dos modelos.

Nas funcionalidades de *codecs*, existe variabilidade ao nível do tipo de algoritmo de compressão/descompressão de áudio/vídeo utilizado: *codec* MJPEG, *codec* MPEG-4 e *codec* H.264. Cada um dos *codecs* também apresentam variabilidade ao nível da compressão/descompressão de áudio/vídeo. Por exemplo, o *codec* de imagem JPEG, apresenta variabilidade ao nível: do tipo de compressão (*Baseline*, Sequencial e Hierárquico), do código entrópico (*Huffman* ou Aritmético), do espaço de cor (YUV, RGB, etc), entre outros (Pennebaker and Mitchell, 1992).

As funcionalidades de processamento apresentam variabilidade ao nível do algoritmo de processamento de imagem utilizado: deteção, classificação e acompanhamento (*tracking*) de objetos, acompanhamento do objeto através de várias câmaras (*tracking* multi-câmara) e deteção de comportamentos anómalos. Em cada um dos algoritmos de processamento de imagem, também existe variabilidade ao nível das funcionalidades internas.

Nas funcionalidades de gravação, existe variabilidade ao nível do tipo de formato de áudio/vídeo utilizado na gravação: formato MOV (*QuickTime Movie*), formato AVI (*Audio Video Interleave*), formato WAV (*Waveform Audio File*) formato MP4 (*Moving Picture Experts Group* 4) e formato proprietário. Em cada um dos formatos, também existe variabilidade ao nível das funcionalidades internas.

As funcionalidades do sistema operativo apresentam variabilidade ao nível do sistema operativo onde o sistema de videovigilância vai ser executado: sistema operativo Windows, sistema operativo Linux e um sistema operativo embebido (Windows ou Linux). O mesmo acontece com as funcionalidades da arquitetura, onde existem várias arquiteturas de *hardware* passíveis de correr o sistema de videovigilância: arquitetura x86, arquitetura AMD, arquitetura PowerPC e arquitetura ARM/DSP da Texas.

Nas funcionalidades de servidor, existe variabilidade ao nível do protocolo utilizado

para envio remoto de áudio/vídeo: protocolo RTSP/RTP (*Real Time Streaming Protocol*/*Real-time Transport Protocol*), protocolo HTTP (*Hypertext Transfer Protocol*), protocolo RTMP (Real Time Messaging Protocol) e protocolo proprietário. Em cada um dos protocolos, também existe variabilidade ao nível das funcionalidades internas. A título de exemplo, o protocolo RTSP/RTP pode enviar ou receber áudio/vídeo utilizando o protocolo da camada de transporte TCP (*Transmission Control Protocol*) ou UDP (*User Datagram Protocol*).

Para obviar ou reduzir o esforço de engenharia necessário para a gestão destas várias classes de variabilidades, têm surgido *middlewares* e *frameworks* multimédia *standards*, que consequentemente aumentam a produtividade e diminuem a pressão sobre o *time-to-market* durante o desenvolvimento de sistemas de videovigilância. Embora os *middlewares standards* e as *frameworks* multimédia sejam bastante flexíveis e suportem um vasto número de funcionalidades que podem ser aplicadas em vários domínios de aplicação, produzem um impacto negativo na performance dos sistemas de videovigilância. Especialmente, quando estes são compilados para serem executados em sistemas embebidos. Isso acontece muito devido ao facto do objetivo dos *middlewares standards* e *frameworks* multimédia fornecerem um suporte genérico a casos comuns, não explorando a especificidade das aplicações em estudo e de utilizarem técnicas de programação dinâmicas para gerir a variabilidade do código do sistema.

Nesta tese é apresentada uma nova abordagem para o desenvolvimento de uma *framework* generativa para sistemas de videovigilância. Na fase de modelação da *framework* generativa, é usada uma combinação de tecnologia e técnicas tais como, *Software Product Line* (SPL) Pohl et al. (2005), *Model Driven Development* (MDD) (Stahl et al., 2006), *template metaprogramming* (TMP) (Czarnecki and Eisenecker, 2000) e *Agile* (Cockburn, 2002), de modo a gerir a variabilidade existente ao nível das funcionalidades do sistema. Na fase da implementação das funcionalidades, são utilizados componentes totalmente configuráveis, desenvolvidos em C++ *template metaprogramming*, de modo a gerir a variabilidade do código do sistema.

Por fim, é apresentado um ambiente de desenvolvimento de sistemas de videovigilância inteligentes, que utiliza o perfil da aplicação para guiar no processo de desenvolvimento da mesma. Assim, ao utilizar o perfil da aplicação, no ato da compilação da *framework* generativa de videovigilância, é produzido um sistema totalmente otimizado para a aplicação em estudo e para um *hardware* dedicado.

1.1 Motivações

O aumento da criminalidade e a possibilidade da ocorrência de eventos que possam por em causa a segurança das pessoas e os seus bens, fizeram com que o número de sistemas de videovigilância instalados em espaços públicos e privados, aumentasse de forma considerável. A generalização dos sistemas de videovigilância deve-se, essencialmente, a dois fatores que estão relacionados com as funções que estes desempenham como agentes de segurança: a possibilidade de agir de forma automática, perante a deteção de um evento anómalo (agente proactivo) e a possibilidade de consultar os eventos anómalos, após a sua ocorrência (agente reativo).

A necessidade de monitorizar espaços de grandes dimensões, tanto ao nível espacial como geográfico, levou a comunidade científica e a indústria a estudar novas abordagens de desenvolvimento, onde os sistemas de videovigilância são vistos como uma rede de sensores distribuída. Como a maioria dos sistemas de videovigilância são desenvolvidos de forma *ad hoc*, a sua integração com outros sistemas é uma tarefa difícil, ou até mesmo impossível.

Constata-se assim, que o uso de arquiteturas para sistemas de videovigilância inteligentes, sob a forma de *middleware* e *frameworks* especializados, baseados em paradigmas multiagentes é uma necessidade real, para que estes tipos de sistemas possam ser integrados em ambientes distribuídos. Desta forma, é possível gerir a complexidade e heterogeneidade que caracterizam este tipo de sistemas, e fornecer um apoio genérico às várias arquiteturas de videovigilância existentes.

Como resposta a esta necessidade, no âmbito desta tese propõem-se o desenvolvimento de: (i) uma *framework* generativa de videovigilância totalmente reconfigurável, que promove a combinação e interoperabilidade entre técnicas e tecnologias de *Software Product Line* (SPL), *Model Driven Development* (MDD), *Agile* e C++ *template metaprogramming*, para gerir a variabilidade das funcionalidades do sistema. Assim, torna-se mais simples desenvolver e configurar aplicações de videovigilância, para serem executadas em sistemas embebidos e totalmente adaptadas para ambientes distribuídos.

1.2 Objetivos e Contribuições

Os objetivos desta tese de doutoramento consistiram: (i) na implementação de uma *framework* generativa de videovigilância, totalmente reconfigurável e que permite gerir de forma eficaz o elevado grau de variabilidade que caraterizam os sistemas de videovigilância de terceira geração, sem pôr em causa o desempenho do mesmo e (ii) desenvolver um ambiente de desenvolvimento de sistemas de videovigilância inteligentes, que utiliza o perfil da aplicação para guiar o processo de desenvolvimento da mesma, fazendo que no ato da compilação seja gerado um ficheiro executável totalmente otimizado para a aplicação alvo.

As principais contribuições desta tese de doutoramento são:

i. Desenvolvimento de uma *framework* generativa de videovigilância totalmente reconfigurável, onde: (i) na fase de modelação da *framework*, promoveu-se a combinação e interoperabilidade entre tecnologias de *Software Product Line* (SPL), *Model Driven Development* (MDD) e *Agile*, de modo a gerir a variabilidade existente ao nível das funcionalidades do sistema e (ii) no processo de implementação das funcionalidades, foram utilizados componentes totalmente configuráveis, desenvolvidos em C++ *template metaprogramming*, de modo a gerir a variabilidade do código do sistema e promover a reutilização os mesmos (Cardoso et al., 2011, 2012a,b,c,d,e,f).

ii. Desenvolvimento de um ambiente de desenvolvimento de sistemas de videovigilância inteligentes, que utiliza o perfil da aplicação para guiar no processo de desenvolvimento da mesma (Cardoso et al., 2012c).

iii. Desenvolvimento da ferramenta de modelação TMPxML *Template Markup Language*, que utiliza modelos para simplificar o processo de desenvolvimento de código C++ *template metaprogramming*. A ferramenta proposta, utiliza a linguagem TMPxML, que tem como base a linguagem XML. Assim, é possível utilizar ferramentas *mainstream* para validação e transformação do código TMPxML para código TMP (Cardoso et al., 2012g).

1.3 Organização da Tese

O estado de arte das matérias abordadas nesta tese de doutoramento apesar de ter como base principal o capítulo 2, encontra-se dividido pelos vários capítulos da

tese. Assim, em cada capítulo desta tese de doutoramento é feito o levantamento do estado de arte das matérias envolventes.

Esta tese está dividida em oito capítulos e dois anexos. No primeiro capítulo, é feita uma breve introdução do problema a resolver, são apresentadas as motivações que levaram ao desenvolvimento deste projeto científico, os objetivos a alcançar e as contribuições deste trabalho para a comunidade científica e para a indústria.

O segundo capítulo, aborda o "estado da arte", onde se descreve a evolução dos sistemas de videovigilância ao longo dos últimos anos e os problemas associados à forma como é feita a gestão da variabilidade das funcionalidades dos sistemas de videovigilância atuais. O capítulo seguinte, incide sobre o impacto que algumas das funcionalidades da linguagem orientada a objetos C++ produzem no desempenho dos sistemas, especialmente quando estes são compilados para serem executados em sistemas embebidos.

O quarto capítulo, descreve a forma como a *framework* multimédia *DirectShow* é implementada e o impacto negativo que algumas das funcionalidades da *framework* produzem em plataformas embebidas. O capítulo cinco, apresenta as principais *frameworks* que utilizam *template metaprogramming* para gerir a variabilidade do código de um sistema sem pôr em causa o desempenho do mesmo.

O capítulo que se segue, descreve o processo de análise, desenho e implementação da *framework* generativa para sistemas de videovigilância. No capítulo sete são apresentados os resultados experimentais. Por fim, no capítulo oito são apresentadas as conclusões e o trabalho futuro.

No anexo A, é descrito o processo de desenvolvimento de um novo filtro de entrada, a adicionar à *framework* generativa de videovigilância. Por fim, no anexo B, é apresentado o XML *schema* da linguagem de modelação TMPxML, que permite gerar código TMP de uma forma mais simples e eficaz.

Capítulo 2

Estado da Arte

2.1 Introdução

Neste capítulo, é apresentado o estado da arte das áreas que estão diretamente relacionadas com o trabalho desenvolvido nesta tese de doutoramento. Assim, numa primeira fase, são apresentados os três tipos de sistemas de videovigilância, onde se destacam as principais diferenças entre eles e as necessidades/avanços tecnológicos que justificaram a evolução dos sistemas de videovigilância.

Seguidamente, é descrita a arquitetura dos atuais sistemas de videovigilância, os principais problemas associados à forma como estes são desenhados/implementados e a relação que existe entre duas das principais métricas que caracterizam este tipo de sistemas: a variabilidade e o desempenho.

Devido à necessidade de gerir de forma eficaz a variabilidade dos sistemas de videovigilância, na secção seguinte, é descrita a tecnologia *Software Product Line* (SPL). Nesta secção, também são apresentados os vários mecanismos da linguagem de programação C++ que permitem gerir a variabilidade do código, onde são apresentadas as vantagens e as desvantagens de cada um deles.

Finalmente, é feita uma breve descrição dos "trabalhos relacionados" com área de investigação desta tese de doutoramento.

2.2 Evolução dos Sistemas de Videovigilância

Os primeiros sistemas de videovigilância surgiram à cerca de 25 anos (Axis). Na altura os sistemas eram totalmente analógicos, constituídos apenas por um grupo restrito de câmaras, que estavam ligadas a monitores num circuito fechado de televisão, denominado de CCTV (*Closed-Circuit Television Cameras*) (Valera and Velastin, 2005; Raty, 2010).

Devido aos avanços tecnológicos e à necessidade de tornar os sistemas de videovigilância cada vez mais autónomos e robustos, houve a necessidade de iniciar o processo de transformação da tecnologia analógica para a tecnologia digital (Duque, 2009). O uso de tecnologia digital em sistemas de videovigilância, permitiu que os sistemas fossem mais flexíveis ao nível da instalação e da manutenção, bem como a possibilidade de executar tarefas automáticas de processamento, até então impossíveis de serem executadas (Ahmad et al., 2008).

O processo de transformação da tecnologia analógica para a tecnologia digital foi gradual, dando origem a 3 gerações de sistemas de videovigilância: os sistemas de videovigilância de 1ª geração totalmente analógicos, os sistemas de videovigilância de 2ª geração híbridos e os sistemas de videovigilância de 3ª geração totalmente digitais (Valera and Velastin, 2005; Raty, 2010).

2.2.1 Sistemas de Videovigilância de 1ª Geração

Tal como foi referido anteriormente, os sistemas de videovigilância de 1ª geração (1960-1980) (Raty, 2010), eram sistemas centralizados totalmente baseados em sinais analógicos, constituídos por um conjunto restrito de câmaras de vídeo analógicas, que permitiam a um operador a visualização das imagens numa central de controlo. Na central de controlo, era também possível fazer a gravação do sinal analógico recebido de cada uma das câmaras em cassetes analógicas num VCR (*Video Cassette Recorder*) (Valera and Velastin, 2005).

No sistema de videovigilância de 1ª geração da figura 2.1, as câmaras analógicas eram ligadas às entradas de um multiplexador através de cabos coaxiais. As saídas do multiplexador eram ligadas a um monitor analógico, que permitia visualizar periodicamente as imagens de cada uma das câmaras de forma sequencial, e a um VCR, que gravava as imagens das mesmas em cassetes analógicas. Basicamente,

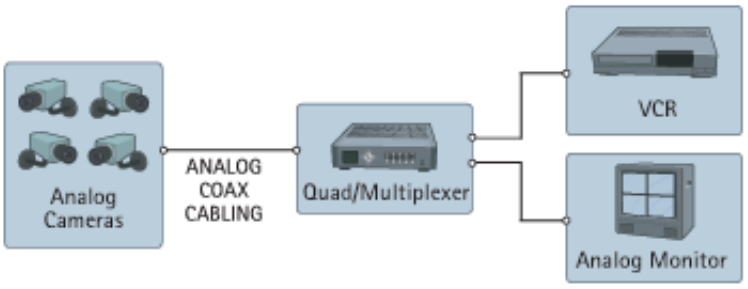


Figura 2.1: Exemplo de um sistema de videovigilância de 1ª geração. Imagem extraída de (Axis).

os sistemas de videovigilância de 1ª geração podem ser vistos como uma extensão do olho humano do ponto de vista espacial.

As principais vantagens da utilização deste tipo de sistemas são o bom desempenho e a maturidade da tecnologia utilizada. A principal desvantagem prende-se com o facto de se utilizarem técnicas analógicas para gravação e distribuição de imagens (Valera and Velastin, 2005).

2.2.2 Sistemas de Videovigilância de 2ª Geração

Os sistemas de videovigilância de 2ª geração (1980-2000) (Raty, 2010), eram sistemas híbridos, isto é, sistemas que utilizavam tecnologia analógica e tecnologia digital. Como base utilizavam a maturidade da tecnologia analógica, e para suprimir alguns dos problemas associados aos sistemas de videovigilância de 1ª geração, era utilizada tecnologia digital (Valera and Velastin, 2005).

A transição do analógico para o digital ocorreu nos sistemas de aquisição de sinal, processamento de imagem e armazenamento de imagem. Assim, apareceram os primeiros DVRs (*Digital Video Recorder*) que recebiam os sinais analógicos das câmaras, transformavam o sinal analógico em sinal digital, de forma a que este pudesse ser processado por algoritmos de processamento de imagem não muito complexos e gravado em discos duros.

Um dos pontos onde a transição do analógico para o digital foi gradual, foi na transmissão do sinal. Assim, ao converter o sinal de analógico para digital e posteriormente de digital para analógico, era possível aumentar de forma significativa a distância entre as câmaras e o ponto recetor.

A figura 2.2 apresenta um exemplo de um sistema de videovigilância de 2ª geração:

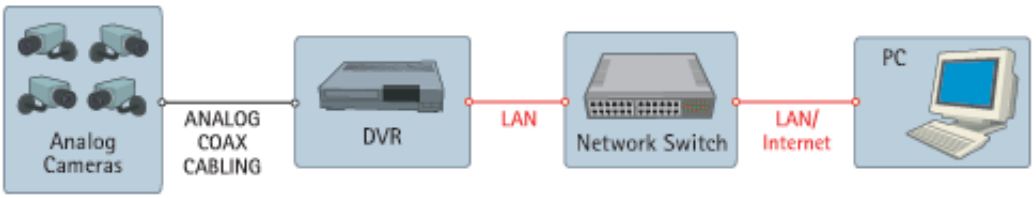


Figura 2.2: Exemplo de um sistema de videovigilância de 2ª geração. Imagem extraída de (Axis).

Tal como ilustra a figura 2.2, as câmaras analógicas eram ligadas às entradas de um DVR através de cabos coaxiais. O DVR fazia a conversão do sinal das câmaras do formato analógico para o formato digital, de forma a que estas pudessem ser processadas e gravadas em disco, para posterior consulta, e transmitidas via rede para visualização remota. Para visualizar as imagens eram utilizados PCs em vez de monitores analógicos.

A principal vantagem da utilização deste tipo de sistemas relativamente ao seu antecessor, é o aumento da eficiência, a possibilidade de executar alguns algoritmos de processamento de imagem e a possibilidade de haver uma maior distância entre as câmaras e o ponto de monitorização. A principal desvantagem prende-se com o facto de os algoritmos de processamento de imagem não serem robustos (Valera and Velastin, 2005).

2.2.3 Sistemas de Videovigilância de 3ª Geração

Os sistemas de videovigilância de 3ª geração surgiram a partir do ano 2000 (Raty, 2010) e utilizam apenas tecnologia digital. A migração da tecnologia analógica para a tecnologia digital, permitiu que os sistemas fossem distribuídos, sendo constituídos por um elevado número de câmaras digitais, que estão ligadas a uma rede de área local. Desta forma, é possível utilizar as infraestruturas existentes, reduzindo assim os custos de instalação associado às mesmas (Valera and Velastin, 2005).

As câmaras digitais (câmaras IP), não só são responsáveis pela a aquisição do sinal de vídeo no formato analógico, como também são responsáveis pela conversão do mesmo para o formato digital. Devido ao aumento considerável da capacidade de processamento das câmaras, é possível executar algoritmos de processamento de imagem na própria câmara e comprimir as imagens de forma a que estas possam ser transmitidas pela rede, minimizando a largura de banda requerida para a transmissão das mesmas. Desta forma, o processamento passa a ser distribuído

pelas várias câmaras que estão espalhadas pela rede, fazendo com que os sistemas sejam escaláveis e robustos (Chao and Jun, 2008).

A figura 2.3 apresenta um exemplo de um sistema de videovigilância de 3ª geração:

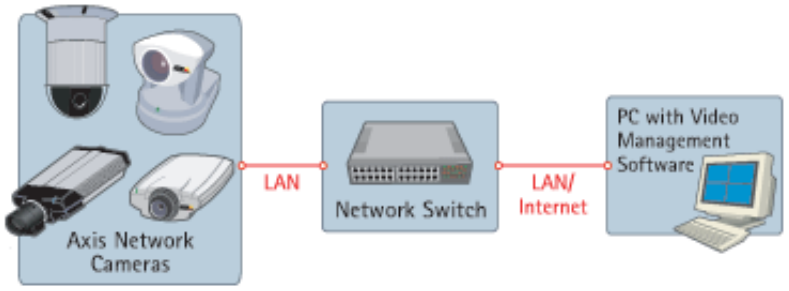


Figura 2.3: Exemplo de um sistema de videovigilância de 3ª geração. Imagem extraída de (Axis).

Tal como ilustra a figura 2.3, o sistema é constituído por várias câmaras IP que estão ligadas a uma rede de área local. Como as câmaras são responsáveis pelo processamento, compressão e transmissão das imagens, é possível visualizar e gerir as imagens através de vários pontos de monitorização que estão espelhados pela rede (normalmente através de um *software* de videovigilância instalado num PC).

Este tipo de sistemas apresenta como principais vantagens a possibilidade de obter mais informações sobre a cena a ser monitorizada, resultado da combinação da informação obtida das várias câmaras, a escalabilidade, flexibilidade e robustez. Como principais desvantagens destacam-se a dificuldade existente na distribuição da informação, nas metodologias de desenho dos sistemas e as diferentes tecnologias e plataformas utilizadas no desenvolvimento das câmaras IP (Valera and Velastin, 2005).

2.3 Arquitetura Atual dos Sistemas de Videovigilância

Embora seja possível encontrar alguns sistemas de videovigilância de 2ª geração na indústria, a maioria dos sistemas de videovigilância atuais são de 3ª geração. Assim, é comum ver instalações de sistemas de videovigilância que utilizam arquiteturas distribuídas, onde cada um dos nodos de aquisição/processamento que constituem o sistema, é responsável pela realização de uma tarefa específica do do-

mínio de execução global do sistema de videovigilância (Wang et al., 2012a; Senst et al., 2011; Wang et al., 2012b).

Devido ao elevado nível de variabilidade das funcionalidades que necessitam de ser suportadas e geridas pelos sistemas de videovigilância de 3ª geração, a maioria dos sistemas utilizam arquiteturas distribuídas e modulares baseadas em filtros. Normalmente, como base de implementação utilizam conceitos implementados nas *frameworks* multimédia *DirectShow* (Pesce, 2002; Lin and Tang, 2011) e *GStreamer* (GStreamer; Lanoe and Senn, 2012). Exemplos de sistemas de videovigilância de 3ª geração desenvolvidos pela comunidade científica e pela indústria são: PRISMATICA (Valera and Velastin, 2005), ADVISOR (Valera and Velastin, 2005) e VSAM (Valera and Velastin, 2005).

A figura 2.4 apresenta a arquitetura do sistema de videovigilância PRISMATICA:

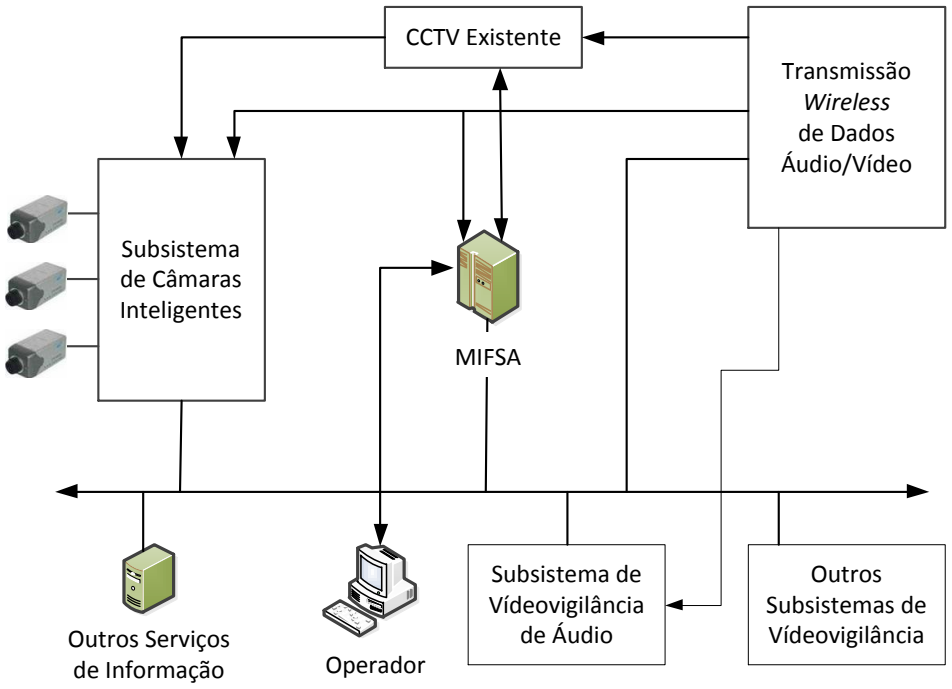


Figura 2.4: Arquitetura do sistema de videovigilância PRISMATICA. Imagem adaptada de (Lo et al., 2003)

O software de videovigilância PRISMATICA (Velastin et al., 2004, 2005) (*Proactive Integrated Systems for Security Management by Technological Institutional and Communication Assistance*) foi desenvolvido com o intuito de monitorizar os aspetos sociais, éticos, organizacionais e técnicos da segurança nos transportes públicos. O sistema utiliza uma arquitetura distribuída constituída por um conjunto de sensores inteligentes de rede que enviam e recebem informações para um mó-

dulo que implementa a central de rede, ao qual se dá o nome de 'MIPSA'. Desta forma, a central de rede 'MIPSA' coordena a atividade dos dispositivos, recebe e envia dados para os mesmos e fornece uma interface gráfica para que um operador possa visualizar eventos e atuar sobre o sistema.

2.3.1 Principal Problema dos Atuais Sistemas de Videovigilância

Tal como foi referido anteriormente, os sistemas de videovigilância de 3ª geração utilizam arquiteturas distribuídas. Os sistemas são constituídos por nós inteligentes que estão espalhados pela rede, podendo um nó ser uma simples câmara IP, um subsistema constituído por várias câmaras IP ou até mesmo um sistema de CCTV já existente. Para processar os eventos enviados pelos nós inteligentes e enviar comandos de controlo para os nós inteligentes, são instaladas unidades de controlo automáticas na rede (Detmold et al., 2006b).

Como cada um dos nós inteligentes podem utilizar funcionalidades e tecnologias diferentes, a gestão dos vários dispositivos torna-se uma tarefa muito complexa. A figura 2.5 apresenta a variabilidade existente ao nível das funcionalidades e tecnologias utilizadas pelos vários nós inteligentes que constituem os sistemas de videovigilância de 3ª geração:

Tal como ilustra o diagrama da figura 2.5, os nós inteligentes utilizam funcionalidades que apresentam um elevado nível variabilidade: variabilidade ao nível das funcionalidades de entrada (Sedky et al., 2005; Desurmont et al., 2004; Ziliani, 2005), variabilidade ao nível das funcionalidades de *codecs* (Ziliani, 2005; Wijnhoven et al., 2004), variabilidade ao nível das funcionalidades de gravação, variabilidade ao nível das funcionalidades de processamento, variabilidade ao nível da arquitetura (Wijnhoven et al., 2004; Hampapur et al., 2003), variabilidade ao nível dos sistemas operativos e variabilidade ao nível das funcionalidades do servidor.

Para resolver o problema relacionado com a gestão da variabilidade das funcionalidades dos sistemas de videovigilância de 3ª geração, a maioria das empresas de desenvolvimento utilizam técnicas de programação dinâmicas baseadas em *plugins* (Garcia-Valls et al., 2010), que utilizam de forma intensiva interfaces e funções virtuais. O uso intensivo de interfaces e funções virtuais produzem um impacto negativo no desempenho do sistema, devido essencialmente a três fatores: as in-



Figura 2.5: Variabilidade existente ao nível das funcionalidades e tecnologias utilizadas nos vários nós inteligentes que constituem os sistemas de videovigilância de 3ª geração.

direções criadas no código executável final do sistema provocadas pelas funções virtuais (Driesen and Hölzle, 1996), o uso de *plugins* afeta de forma negativa a linearidade do código do sistema, fazendo com que aumente a probabilidade da ocorrência de *cache misses* e *page faults* (Cardoso et al., 2011; Plauger, 1997; Meyers), e a impossibilidade de explorar a sinergias que possam existir entre os vários filtros que compõem o pipeline de processamento de cada um dos nós do sistema (Cardoso et al., 2012c; Saks, 2002; per).

A figura 2.6 apresenta a relação que existe entre a gestão da variabilidade e o desempenho de um sistema de videovigilância de 3ª geração:

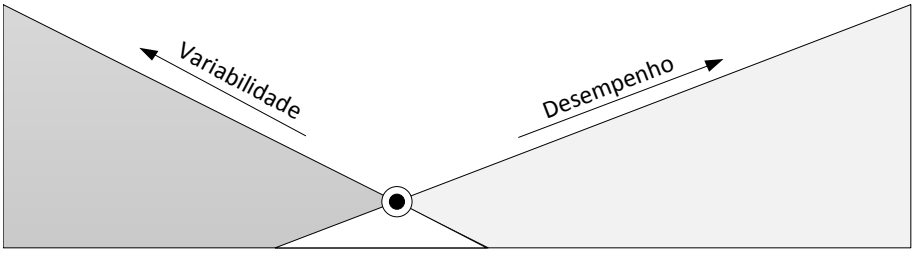


Figura 2.6: Relação que existe entre a gestão da variabilidade e o desempenho de um sistema de videovigilância de 3ª geração.

Pelo gráfico conclui-se que quanto maior for a variabilidade do sistema a gerir, me-

nor é o desempenho do mesmo. De forma inversa, se o objetivo for obter um maior desempenho do sistema, menor deve ser a variabilidade a gerir pelo sistema. Logo, terá sempre que haver um bom compromisso entre a gestão da variabilidade e o desempenho do sistema, através de um bom balanceamento entre as duas métricas. Para isso, é proposto nesta tese o uso das técnicas de engenharia de *Software Product Line* no desenvolvimento e manutenção de sistemas de videovigilância de 3ª geração (Cardoso et al., 2012c).

2.4 *Software Product Line*

Software Product Line (SPL) (Clements and Northrop, 2001; Weiss and Lai, 1999; Pohl et al., 2005) engloba um conjunto de técnicas de engenharia, que permitem explorar as funcionalidades que são comuns a uma família de produtos, de forma a que seja possível criar um novo produto a partir da uma base comum. Como o objetivo do SPL é desenvolver produtos diferenciados através da mesma linha de produção, o foco do desenvolvimento não está direcionado apenas para um produto específico, mas sim para uma família de produtos do domínio em questão (Withey, 1996), neste caso específico a família de produtos do domínio da videovigilância.

Assim, numa primeira fase é necessário identificar e implementar os componentes que são comuns e os componentes que são variáveis na linha de produção. Desta forma, é possível desenvolver componentes que são reutilizáveis na produção de um produto específico da linha de produção ("*developing for reuse*"). Por fim, para produzir um produto específico da linha de produção, são selecionados de um repositório de componentes os componentes necessários para obter o produto em questão ("*developing with reuse*") (Karlsson, 1995; Czarnecki and Eisenecker, 2000).

As principais vantagens de utilizar SPL no desenvolvimento do *software* são: a redução do *time-to-market*, o aumento da qualidade dos produtos e o aumento da produtividade (Krueger, 1992; Clements and Northrop, 2001; Lim, 1994). O *time-to-market* é reduzido porque sempre que é necessário implementar um novo produto, apenas é necessário implementar os componentes do produto que são variáveis. A qualidade dos produtos aumenta devido ao facto de se utilizarem componentes já devidamente testados na produção de novos produtos. Por fim, devido à redução do esforço e dos custos necessários para desenvolver e fazer a manutenção da mesma família de produtos, a produtividade aumenta.

2.4.1 Conceitos e Terminologia

A principal diferença que existe no processo de desenvolvimento de um sistema orientado para a produção de um único produto e o desenvolvimento de um sistema orientado para a produção de uma família de produtos, reside no facto de no segundo caso, existir uma separação lógica entre o processo de análise, desenho e implementação de todos os componentes que constituem a família de produtos ("*developing for reuse*") e o processo de seleção e configuração dos componentes que constituem um produto específico ("*developing with reuse*").

Assim, ao desenvolver um sistema utilizando as técnicas de engenharia do SPL, o processo de desenvolvimento é dividido em duas fases: o *domínio de engenharia* ("*developing for reuse*") e a *engenharia da aplicação* ("*developing with reuse*").

Domínio de Engenharia

O domínio de engenharia engloba um conjunto de etapas que consistem: (i) num conjunto de especificações e funcionalidades que um especialista do domínio necessita de utilizar para expressar as suas necessidades (*espaço do problema*), (ii) na implementação dos componentes que permitem obter todas as configurações possíveis do espaço do problema (*espaço da solução*) e (iii) no mapeamento entre o espaço do problema e o espaço da solução (*configuração do conhecimento*).

A figura 2.7 apresenta as etapas do estágio do *domínio da engenharia* do SPL:

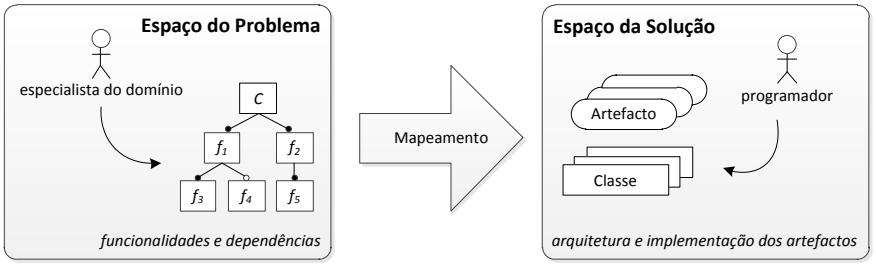


Figura 2.7: Etapas do estágio do domínio da engenharia do SPL. Imagem adaptada de (Lohmann, 2009).

Da figura 2.7, conclui-se que para cada uma das instâncias do espaço do problema (*problema específico*), existe uma instância no espaço da solução (*solução específica*). Ao nível da modelação, um especialista do domínio específica a variabilidade do espaço do problema utilizando modelos formais das funcionalidades e as suas dependências. O programador implementa essa variabilidade no espaço da solução

através de componentes totalmente configuráveis e fornece um mapeamento formal entre os componentes e as funcionalidades.

Engenharia da Aplicação

A engenharia da aplicação engloba as seguintes etapas: (i) a seleção das funcionalidades pretendidas para obter a configuração pretendida (*problema específico*) e (ii) a geração automática do ficheiro executável totalmente otimizado para o fim proposto (*solução específica*).

A figura 2.8 apresenta as etapas do estágio da *engenharia da aplicação* do SPL:

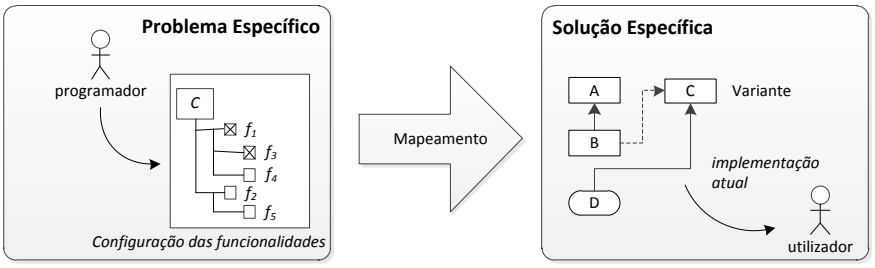


Figura 2.8: Etapas do estágio da engenharia da aplicação do SPL. Imagem adaptada de (Lohmann, 2009).

Da figura 2.8, conclui-se que ao nível da instanciação o programador da aplicação especifica as propriedades pretendidas ao selecionar as funcionalidades (*configuração*). Ao selecionar as funcionalidades pretendidas, é feito um mapeamento para gerar de forma automática os componentes otimizados para satisfazer as necessidades do programador (*variante*).

2.4.2 Especificação do Espaço do Problema

Tal como foi referido anteriormente, o espaço do problema é constituído pelas especificações do sistema, que são estabelecidas durante a análise do domínio e a fase de engenharia dos requisitos. Assim, são identificadas as funcionalidades que são comuns e as funcionalidades que são variáveis no domínio, através da modelação das funcionalidades (Czarnecki et al., 2005).

A modelação das funcionalidades foi proposta como uma parte do *Feature-Oriented Domain Analysis* (FODA) (Kang et al., 1990), e desde aí, é aplicada em vários domínios. A modelação das funcionalidades é representada visualmente através do

diagrama de funcionalidades, que representa as funcionalidades do sistema organizadas de forma hierárquica e a relação entre elas, onde o nó raiz representa o conceito e os nós descendentes representam as funcionalidades do conceito (Czarnecki and Eisenecker, 2000).

A figura 2.9 apresenta os tipos de funcionalidades mais utilizadas num diagrama de funcionalidades:

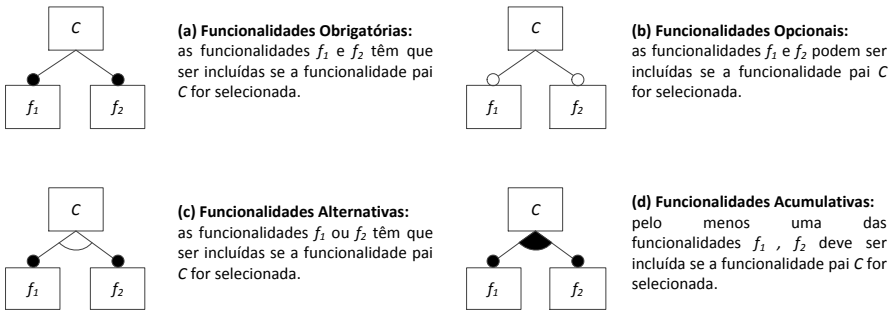


Figura 2.9: Tipos de funcionalidades mais utilizadas num diagrama de funcionalidades. Imagens adaptadas de (Lohmann, 2009).

A figura 2.9 apresenta quatro tipos de funcionalidades:

- *Funcionalidades Obrigatórias*: o sistema é constituído por determinadas funcionalidades. Este tipo de funcionalidades é representado por uma reta que termina com um círculo com preenchimento.
- *Funcionalidades Opcionais*: o sistema pode ser constituído por determinadas funcionalidades. Este tipo de funcionalidades é representado por uma reta que termina com um círculo sem preenchimento.
- *Funcionalidades Alternativas*: o sistema pode ser constituído apenas por uma funcionalidade ao mesmo tempo. Este tipo de funcionalidades é representado por uma reta com um arco.
- *Funcionalidades Acumulativas*: o sistema pode ser constituído pela combinação das funcionalidades. Este tipo de funcionalidades é representado por uma reta com um arco preenchido.

A principal vantagem de se utilizarem diagramas de funcionalidades é a possibilidade de modular as funcionalidades do sistema sem ter em consideração o tipo de mecanismo de gestão da variabilidade utilizado ao nível da implementação, tais como, herança, *templates* ou compilação condicional (Czarnecki and Eisenecker, 2000).

A figura 2.10 apresenta um exemplo típico de um diagrama de funcionalidades (Czarnecki and Eisenecker, 1999):

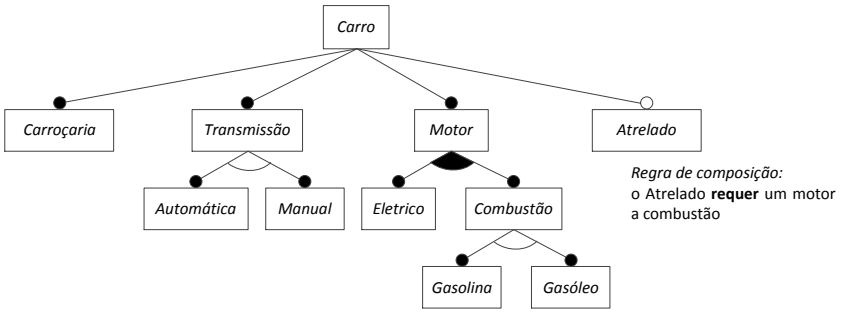


Figura 2.10: Exemplo típico de um diagrama de funcionalidades. Diagrama extraído de (Czarnecki and Eisenecker, 1999)

O diagrama de funcionalidades da figura 2.10 representa o conceito *Carro*, que é constituído por três funcionalidades obrigatórias, a funcionalidade *Carroçaria*, a funcionalidade *Transmissão* e a funcionalidade *Motor*; e a funcionalidade opcional *Atrelado*.

A funcionalidade *Transmissão* é constituída por duas funcionalidades alternativas: a funcionalidade *Automática* e a funcionalidade *Manual*. Isto é, um carro ou tem transmissão automática ou transmissão manual.

A funcionalidade *Motor* é constituída por duas funcionalidades acumulativas: a funcionalidade *Elétrico* e a funcionalidade *Combustão*. Assim, o motor do carro pode ser elétrico, a combustão ou híbrido.

Finalmente, a funcionalidade *Combustão* é constituída por duas funcionalidades alternativas: a funcionalidade *Gasolina* e a funcionalidade *Gasóleo*. Isto é, um motor a combustão ou é a gasolina ou a gasóleo.

De realçar, que o diagrama apresenta uma regra de composição que indica que só é possível rebocar um atrelado se o motor do carro for a combustão.

2.4.3 Implementação do Espaço da Solução

O objetivo do espaço da solução é implementar os componentes necessários para obter todas as configurações possíveis do espaço do problema. Assim, como em SPL todos os produtos da família são constituídos por funcionalidades comuns, são utilizadas classes para implementar o código dos componentes que existem em

todas as configurações. Para implementar a variabilidade, isto é, aquilo que realmente diferencia um produto dos outros produtos da família, são implementados artefactos onde para gerir a variabilidade podem ser utilizados vários mecanismos de gestão da variabilidade (Clements and Northrop, 2001).

Mecanismos de Gestão da Variabilidade

Tal como foi referido anteriormente, ao utilizar as técnicas de engenharia SPL, o foco do desenvolvimento em vez de ser orientado para um produto específico, é orientado para uma família de produtos do domínio em questão, neste caso específico a família de produtos do domínio da videovigilância.

Assim, na fase de implementação dos componentes deve-se ter em consideração o facto de estes serem implementados de forma genérica, satisfazendo as necessidades de uma família de produtos em particular e possibilitando que estes possam ser reutilizados facilmente no contexto da família (Pohl et al., 2005; Czarnecki and Eisenecker, 2000).

Como um dos principais conceitos no desenvolvimento dos componentes para uma família de produtos é a análise das funcionalidades que são comuns a todos os produtos e as funcionalidades que são variáveis (as funcionalidades que diferenciam cada um dos produtos), é necessário identificar o mecanismo de gestão da variabilidade que melhor se adequa para implementar a variabilidade dos produtos (Gacek and Anastasopoules, 2001; Brown et al., 2002).

De seguida, são apresentados os vários mecanismos de gestão da variabilidade que são utilizados pela comunidade científica e pela indústria, e apresentados os pós e os contras de cada um deles (Muthig and Patzke, 2003). Por fim, é apresentado aquele que melhor se adequa para implementar a variabilidade dos componentes do domínio da videovigilância.

Compilação Condicional A compilação condicional permite selecionar as partes do código que são compiladas, enquanto outras são excluídas do processo de compilação. Assim, é possível definir de forma estática as partes do código que são opcionais e as que são exclusivas, através da diretiva de pré-processamento #ifdef.

Ao utilizar compilação condicional é possível implementar a variabilidade com um elevado grau de granularidade, isto é, é possível selecionar apenas uma linha de

código ou múltiplas linhas de código. Aliás, o elevado grau de granularidade é uma das desvantagens da utilização deste tipo de mecanismo de gestão da variabilidade, porque em projetos extensos com um elevado nível de variabilidade, existem muitas diretivas de pré-processamento #ifdef que fazem com que o código seja de difícil compreensão e de difícil manutenção (Lohmann, 2009).

A vantagem da compilação condicional é que não é produzido qualquer tipo de *overhead* em tempo de execução, uma vez que a customização não é efetuada por algum mecanismo de abstração da linguagem de programação, mas sim pelo pré-processador da linguagem C/C++ (Couto et al., 2011). Ao utilizar a compilação condicional como mecanismo de gestão da variabilidade, a variabilidade é resolvida em tempo de compilação.

O código seguinte apresenta um exemplo do uso da compilação condicional para gerir a variabilidade da classe EmailReceiver:

```
class EmailReceiver {
public:
    bool receiveMessage(MSG &msg) {
#ifdef _POP3_DEFINED_
        /* TODO: inserir o código do POP3 */
#endif

#ifdef _IMAP_DEFINED_
        /* TODO: inserir o código do IMAP */
#endif
        return true;
    }
};
```

A classe EmailReceiver permite receber mensagens de um servidor de *email*, utilizando o protocolo POP3 (*Post Office Protocol*) ou o protocolo IMAP (*Internet Message Access Protocol*). Para utilizar o protocolo POP3 é necessário definir a variável de compilação _POP3_DEFINED_. No caso de se utilizar o protocolo IMAP, é necessário definir a variável de compilação _IMAP_DEFINED_.

O código seguinte apresenta um exemplo do uso da classe EmailReceiver, quando esta utiliza o protocolo de rede POP3:

```
#define _POP3_DEFINED_
#undef _IMAP_DEFINED_

MSG msg;
EmailReceiver receiver;
receiver->receiveMessage(&msg);
```

A compilação condicional é um dos mecanismos de gestão da variabilidade mais

utilizados pela comunidade científica e pela indústria. No topo da lista de projetos que utilizam a compilação condicional como mecanismo de gestão da variabilidade, destacam-se: (i) o sistema operativo Linux (Linux; Lotufo et al., 2010; Berger et al., 2010; She et al., 2010), (ii) o sistema operativo tempo-real para sistemas embebidos eCos (eCos; Lohmann, 2009) e (iii) a *framework* multimédia *GStreamer* (GStreamer).

Polimorfismo Dinâmico O polimorfismo dinâmico é um dos mecanismos de gestão da variabilidade mais utilizados pela comunidade científica e por a indústria, quando o código do sistema é implementado numa linguagem de programação orientada a objetos. Aliás, como a maioria dos fabricantes de *software* de videovigilância utilizam a linguagem de programação C++ para implementar os sistemas, o polimorfismo dinâmico é uma prática comum na gestão da variabilidade dos mesmos (Czarnecki and Eisenecker, 2000; Batory et al., 2000).

Para implementar a variabilidade utilizando polimorfismo dinâmico é definida uma classe abstrata (interface) e criadas subclasses que herdam da classe abstrata para implementar os vários pontos de variação, através da reimplementação das funções virtuais definidas na classe abstrata.

Ao utilizar o polimorfismo dinâmico a variabilidade do sistema é resolvida em tempo de execução. Como, normalmente, as subclasses reimplementam apenas as funções virtuais de uma única classe abstrata, o polimorfismo dinâmico está mais orientado para implementar variabilidade do tipo alternativa (XOR) (Muthig and Patzke, 2003).

A principal vantagem do polimorfismo dinâmico deve-se ao facto de ser fácil a sua implementação e manutenção ao nível do código, sendo evidente quais as partes do código do sistema que implementam os vários pontos de variabilidade, visto que cada ponto de variação é encapsulado em classes diferentes.

Como desvantagens apresentam-se duas: (i) como a variabilidade é resolvida dinamicamente em tempo de execução, é introduzido *overhead* no desempenho do sistema devido à indirecção criada pela tabela virtual vptr (Driesen and Hölzle, 1996) e (ii) como o polimorfismo dinâmico apenas permite adicionar um tipo de variabilidade ao mesmo tempo, torna-se muito complexo introduzir variabilidade do tipo acumulativo (OR), sem recorrer a um número elevado de classes e duplicação de código (múltipla herança e múltipla hierarquia de classes), que penalizam de

forma negativa a complexidade e o desempenho dos sistemas (Muthig and Patzke, 2003).

O código seguinte apresenta um exemplo do uso do polimorfismo dinâmico para gerir a variabilidade da classe EmailReceiver:

```
/* definir a classe abstrata */
class EmailReceiver {
public:
    virtual void receiveMessage(MSG &msg) = 0;
};

/* definir a classe que utiliza o protocolo POP3 */
class POP3EmailReceiver : public EmailReceiver {
public:
    void receiveMessage(MSG &msg) {
        /* TODO: inserir o código do POP3 */
    }
};

/* definir a classe que utiliza o protocolo IMAP */
class IMAPEmailReceiver : public EmailReceiver {
public:
    void receiveMessage(MSG &msg) {
        /* TODO: inserir o código do IMAP */
    }
};
```

Numa primeira fase é criada a classe abstrata EmailReceiver que define o método virtual puro receiveMessage. Como para receber o *email* pode-se utilizar o protocolo POP3 ou o protocolo IMAP são definidas duas subclasses da classe abstrata EmailReceiver: a subclasse POP3EmailReceiver e a subclasse IMAPEmailReceiver. O método virtual receiveMessage definido na classe abstrata EmailReceiver, é reimplementado pelas duas subclasses.

O código seguinte apresenta um exemplo do uso da classe EmailReceiver, quando esta utiliza o protocolo de rede POP3:

```
MSG msg;
EmailReceiver *pReceiver = new POP3EmailReceiver();
pReceiver->receiveMessage(msg);
```

Tal como foi referido anteriormente, o polimorfismo dinâmico é um dos mecanismos de gestão da variabilidade mais utilizados pela comunidade científica e pela indústria, principalmente quando o código do sistema é desenvolvido na linguagem de programação C++. Da lista de projetos que utilizam o polimorfismo dinâmico como mecanismo de gestão da variabilidade, destacam-se: (i) a *framework* multimédia *DirectShow* (Pesce, 2002), (ii) o sistema de videovigilância *NUUO* (NUUO)

e (iii) o sistema de videovigilância *GeoVision* (GeoVision).

Polimorfismo Paramétrico O polimorfismo paramétrico permite que seja executado o mesmo código em função do tipo de dados. Em C++ o polimorfismo paramétrico é implementado utilizando *templates*. Tal como a compilação condicional, o polimorfismo paramétrico é resolvido em tempo de compilação (Muthig and Patzke, 2003).

A principal vantagem de utilizar o polimorfismo paramétrico na gestão da variabilidade, prende-se com o facto de não ser introduzido qualquer tipo de *overhead* no desempenho do sistema. Como a variabilidade do código do sistema é resolvida em tempo de compilação, o ficheiro executável gerado pelo compilador é totalmente otimizado para o fim proposto.

Como desvantagem, a comunidade científica e a indústria apontam a complexidade do código das *templates*. Isto é, como as *templates* são um dos conceitos mais avançados da linguagem C++, implica que o programador necessita de dominar esses mesmos conceitos, tornando o desenvolvimento do código uma tarefa mais complexa.

O código seguinte apresenta um exemplo da uso do polimorfismo paramétrico para gerir a variabilidade da classe EmailReceiver:

```
/* define a classe que utiliza o protocolo POP3 */
class POP3EmailReceiver {
public:
        void receiveMessage(MSG &msg) {
                /* TODO: inserir o código do POP3 */
        }
};

/* define a classe que utiliza o protocolo IMAP */
class IMAPEmailReceiver {
public:
        void receiveMessage(MSG &msg) {
                /* TODO: inserir o código do IMAP3 */
        }
};

/* define a classe template EmailReceiver */
template<typename protocol>
class EmailReceiver : public protocol {
        ...
};
```

O código seguinte apresenta um exemplo do uso da classe EmailReceiver, quando esta

utiliza o protocolo de rede POP3:

```
MSG msg;
EmailReceiver<POP3EmailReceiver> receiver;
receiver.receiveMessage(msg);
```

O polimorfismo paramétrico não é muito utilizado pela comunidade científica e pela indústria, como mecanismo de gestão da variabilidade. Um dos exemplos mais conhecidos que utiliza *templates* para gestão da variabilidade do código é a biblioteca STL (*Standard Template Library*) (Meyers, 2001).

Polimorfismo de Operações O polimorfismo de operações, também conhecido por polimorfismo *ad-hoc*, permite que sejam definidas várias funções com o mesmo nome, mas com tipos de dados de entrada diferentes. Assim, no ato da compilação, o compilador chama a função que melhor se adequa ao ambiente (Muthig and Patzke, 2003).

A vantagem da utilização do polimorfismo de operações deve-se ao facto de ser um método de gestão da variabilidade bastante conhecido. Como desvantagem, aponta-se o facto de ser menos importante e poderoso que o polimorfismo universal.

O código seguinte apresenta um exemplo do uso do polimorfismo de operações para gerir a variabilidade da classe EmailReceiver:

```
/* define a estrutura do protocolo POP3 */
struct POP3 { ... };
/* define a estrutura do protocolo IMAP */
struct IMAP { ... };

/* define a classe EmailReceiver */
class EmailReceiver {
public:
    /* define o método que recebe mensagens utilizando o protocolo POP3 */
    void receiveMessage(POP3 protocol, MSG &msg) {
        /* TODO: inserir o código do POP3 */
    }

    /* define o método que recebe mensagens utilizando o protocolo IMAP */
    void receiveMessage(IMAP protocol, MSG &msg) {
        /* TODO: inserir o código do IMAP */
    }
};
```

No código são definidas duas estruturas de dados: a estrutura POP3 e a estrutura IMAP. A estrutura de dados POP3 representa os dados do protocolo de receção de mensagens de *email* POP3. A estrutura de dados IMAP representa os dados do protocolo de receção de mensagens de *email* IMAP. A classe EmailReceiver define

dois métodos com o nome receiveMessage, onde em cada um deles é implementada a funcionalidade dos métodos de receção de mensagens de *email* POP3 e IMAP. No ato da compilação, o tipo de dados do primeiro parâmetro de entrada do método receiveMessage, notifica o compilador qual o método a executar.

O código seguinte apresenta um exemplo do uso da classe EmailReceiver, quando esta utiliza o protocolo de rede POP3:

```
MSG msg;
POP3 protocol;
EmailReceiver receiver;
receiver.receiveMessage(protocol, msg);
```

Como o polimorfismo de operações não é utilizado pela comunidade científica e pela indústria, como mecanismo de gestão da variabilidade, não é apresentado nenhum projeto que o utilize como mecanismo de gestão da variabilidade do código.

C++ *Template Metaprogramming* C++ *template metaprogramming* (TMP) é uma técnica de metaprogramação que utiliza *templates* em C++ para gerar e manipular o código de uma aplicação em tempo de compilação (Czarnecki and Eisenecker, 2000).

Ao desenvolver código utilizando TMP, é possível estender as capacidades do compilador C++, fazendo com que este aja como um interpretador, que emite todas as instruções necessárias para produzir uma configuração estática e otimizada de uma aplicação. Depois do código TMP ser instanciado pelo compilador C++, é produzido código otimizado tal como se este fosse escrito à mão (Abrahams and Gurtovoy, 2004; Gennaro, 2011).

Tal como no caso do polimorfismo paramétrico, a principal vantagem de utilizar TMP na gestão da variabilidade, prende-se com o facto de não ser introduzido qualquer tipo de *overhead* no desempenho do sistema. Mais, como o código TMP faz com que a linguagem de programação C++ seja uma linguagem funcional, é possível estender as funcionalidades do compilador, e obter um nível de otimização de código superior ao obtido pela compilação condicional e polimorfismo paramétrico.

Como desvantagem, apresenta-se o elevado nível da complexidade do código TMP e a forma diferente de pensar os algoritmos (o código TMP está diretamente relacionado com as características das linguagens funcionais).

O código seguinte apresenta um exemplo da uso de TMP para gerir a variabilidade da classe EmailReceiver:

```
/* define a estrutura do protocolo POP3 */
struct POP3 {
        static void receiveMessage(MSG &msg) {
                /* TODO: inserir o código do POP3 */
        }
};

/* define a estrutura do protocolo IMAP */
struct IMAP {
        static void receiveMessage(MSG &msg) {
                /* TODO: inserir o código do IMAP */
        }
};

/* define a classe EmailReceiver */
template<typename T>
class EmailReceiver {
public:
        /* define o método que recebe mensagens */
        void receiveMessage(MSG &msg) {
                mpl::if_<mpl::is_same<T, POP3>,
                        POP3, IMAP>::receiveMessage(msg);
        }
};
```

No código apresentado, são definidas duas estruturas de dados para cada um dos métodos de receção de *email*: a estrutura de dados POP3 e a estrutura de dados IMAP. Em cada uma das estruturas de dados é definido o método estático receiveMessage, que recebe as mensagens de *email*, em função do protocolo pretendido. No método receiveMessage da classe EmailReceiver é utilizada a metafunção if_ da biblioteca *boost* MPL (Gurtovoy and Abrahams; Abrahams and Gurtovoy, 2004), que seleciona a estrutura de dados utilizada para a receção dos *emails*. Na condição de seleção da metafunção if_, é utilizada a metafunção is_same. Caso o tipo de dados T seja do tipo POP3 é executado o método estático receiveMessage da estrutura de dados POP3. Caso contrário, se o tipo de dados T for do tipo IMAP é executado o método estático receiveMessage da estrutura de dados IMAP.

O código seguinte apresenta um exemplo do uso da classe EmailReceiver, quando esta utiliza o protocolo de rede POP3:

```
MSG msg;
POP3 protocol;
EmailReceiver<POP3> receiver;
receiver.receiveMessage(msg);
```

O C++ *template metaprogramming* começa a ocupar um dos lugares de destaque

na comunidade científica e na indústria, como mecanismo de gestão da variabilidade do código em tempo de compilação. Exemplo disso são os seguintes projetos: (i) a biblioteca *Boost* MPL (Gurtovoy and Abrahams; Abrahams and Gurtovoy, 2004), (ii) a biblioteca genérica de imagem GIL (*Generic Image Library*) (Bourdev; Bourdev and Järvi, 2011), (iii) a biblioteca MTL (*Matrix Template Library*) (SimuNova), (iv) o projeto científico que explora a implementação de *device drivers* utilizando TMP (Steup et al., 2010) e (v) o projeto científico que implementa o processamento de uma *BitStream* para sistemas embebidos (Klemm and Fettweis, 2010).

Programação Orientada a Aspetos A programação orientada a aspetos (AOP) é um novo paradigma de programação proposto em (Kiczales et al., 1997). A programação orientada a aspetos (AOP) surge como um complemento à programação orientada a objetos (OOP), permitindo organizar de forma diferente a estrutura de um programa (Czarnecki and Eisenecker, 2000).

Ao estruturar de forma diferente o código de um programa utilizando aspectos, é possível modelar conceitos que estão dispersos pelo código do programa (*cross-cutting concerns*), até então impossíveis de modelar utilizando classes. Assim, o objetivo principal do AOP é permitir ao programador separar o código dos componentes (propriedades do sistema que são totalmente encapsuláveis) do código dos aspectos (propriedades do sistema que não são totalmente encapsuláveis) (Afonso, 2009).

Para produzir o código global do sistema, são fornecidos mecanismos que permitem a composição do código dos componentes e dos aspectos, em função da configuração pretendida. O AspectC++ *weaver* é a ferramenta responsável pela composição do código base do componente em C++ com o código do aspecto, seguindo o modelo de transformação de código *source-to-source* (Afonso, 2009; Alves et al.; Lohmann et al., 2009; Figueiredo et al., 2008).

A vantagem de utilizar AOP na gestão da variabilidade do sistema, prende-se com o facto de ser possível modelar funcionalidades até então impossíveis de modelar com classes em C++. Desta forma, é possível encapsular funcionalidades que estão espalhadas pelo código do sistema, tais como, as funcionalidades de *logging* e de autenticação.

Como desvantagens, apresentam-se duas: a necessidade de utilizar uma ferramenta

externa antes do ato da compilação do código (AspectC++ *weaver*) e necessidade de aprender novos conceitos da linguagem AOP para implementar a variabilidade, tais como, aspect, pointcut e advice (Afonso, 2009).

De seguida é apresentado um exemplo em AOP para gerir a variabilidade da classe EmailReceiver. A classe EmailReceiver em C++ implementa o código base, isto é, o código que é comum a todas as configurações:

```
/* define a classe EmailReceiver */
class EmailReceiver {
public:
        /* define o método que recebe mensagens */
        void receiveMessage(MSG &msg) {
                /* TODO: inserir o código do aspecto respetivo */
        }
};
```

O código variável é implementado em dois aspectos: o aspecto EmailReceiverPOP3 e o aspecto EmailReceiverIMAP:

```
/* define o aspecto do POP3 */
aspect EmailReceiverPOP3 {
        advice execution("% EmailReceiver::receiveMessage(...)") : around() {
                /* TODO: inserir o código do POP3 */
        }
};

/* define o aspecto do IMAP */
aspect EmailReceiverIMAP {
        advice execution("% EmailReceiver::receiveMessage(...)") : around() {
                /* TODO: inserir o código do IMAP */
        }
};
```

Ao executar AspectC++ *weaver*, é injetado no método receiveMessage da classe EmailReceiver, o código definido no *advice* do aspecto que define o método de receção de *email* pretendido. Para definir qual o aspecto a utilizar, por norma executam-se ficheiros de *script* antes de chamar o compilador C++ para compilar o código C++ gerado.

A programação orientada a aspectos (AOP) ocupa um dos lugares de destaque na comunidade científica e na indústria, como mecanismo de gestão da variabilidade do código em tempo de compilação. Exemplo disso são os seguintes projetos científicos: (i) *Aspect-Awareness in the Development of Configurable System Software* Lohmann (2009), (ii) *Operating System Fault Tolerance Support for Real-Time Embedded Applications*) (Afonso, 2009), (iii) *Aspect-Oriented Programming of video-based surveillance systems* (Cardoso et al., 2011), (iv) *Building Custo-*

mizable Middleware using Aspect Oriented Programming (Hunleth, 2002) e (v) *Object-oriented stream programming using aspects* (Wang and Parashar, 2010).

Frame Technology *Frame technology* (FT) é uma linguagem que foi desenvolvida no início dos anos 80 por *Paul G. Bassett* (Bassett, 1987), com o intuito de aumentar a produtividade do *software* através da reutilização adaptativa dos componentes (Bassett, 1996). Os componentes são adaptados de forma automática através de *frames*, que podem ser organizadas de forma hierárquica, possibilitando assim, que *frames* possam ser adaptadas por outras *frames* (Ferreira et al., 2011).

Existem várias implementações da linguagem *Frame technology*, sendo uma das mais conhecidas a linguagem XVCL (*XML-Based Variant Configuration Language*) (XVCL; Paul et al., 2003). O XVCL é uma linguagem de uso genérico que utiliza a notação XML e permite configurar a variabilidade de programas em qualquer tipo de linguagem de programação ou outro tipo de documentos no formato de texto.

A vantagem de utilizar FT para gerir a variabilidade do sistema, esta relacionada o facto de ser possível separar o código de cada uma das funcionalidades variáveis em módulos, sendo este mais percetível e de fácil manutenção. Como desvantagens, apresenta-se a necessidade de aprender uma nova linguagem (XVCL) e a necessidade de utilizar ferramentas externas para gerar o código do sistema com a configuração pretendida.

De seguida é apresentado um exemplo do uso da linguagem XVCL para gerir a variabilidade da classe EmailReceiver. A variabilidade da classe é gerida recorrendo a duas *frames*: a *frame* SPC e a *frame* EmailReceiver. A *frame* SPC define a variabilidade do componente e a *frame* EmailReceiver define o código comum às duas implementações do componente.

O código seguinte apresentada o conteúdo da *frame* SPC que está gravado no ficheiro spc.xvcl:

```
<x−frame name="SPC" outfile="email_receiver.h">
    <set var="POP3_VAR" value="1"/>
    <set var="IMAP_VAR" value="0"/>

    <adapt x−frame="EmailReceiverBase">
        <insert break="POP3_SIGNALS">
            <select option="POP3_VAR">
                <option value="1">
                    /* TODO: inserir o código do POP3 */
                </option>
```

```
            </select>
        </insert>

        <insert break="IMAP_SIGNALS">
            <select option="IMAP_VAR">
                <option value="1">
                    /* TODO: inserir o código do IMAP */
                </option>
            </select>
        </insert>
    </adapt>
</x-frame>
```

O ficheiro spc.xvcl define a *frame* com o nome SPC e o ficheiro email_receiver.h como ficheiro de saída definido na *tag* XML outfile. Na *frame* são definidas duas variáveis de controlo: a variável POP3_VAR e a variável IMAP_VAR. Se o valor da variável POP3_VAR estiver definido a 1, implica que na *frame* será injetado o código do protocolo POP3 para receber mensagens de *email* do servidor. Caso seja o valor da variável IMAP_VAR que esteja definido a 1, implica que na *frame* será injetado o código do protocolo IMAP para receber mensagens de *email* do servidor.

Por fim, é apresentado o código do ficheiro email_receiver_base.xvcl:

```
<x-frame name="EmailReceiver">
    /* define a classe EmailReceiver */
    class EmailReceiver {
    public:
        /* define o método que recebe as mensagens */
        void receiveMessage(MSG &msg) {
            <break name="POP3_SIGNALS"/>
            <break name="IMAP_SIGNALS"/>
        }
    };
</x-frame>
```

O *frame technology* (FT) é pouco utilizado na comunidade científica e na indústria, como mecanismo de gestão da variabilidade do código em tempo de compilação. Exemplo disso são os poucos projetos científicos encontrados: (i) *Modeling and Customization of Fault Tolerant Architecture using Object-Z/XVCL* (Yuan et al., 2006) e (ii) *A hybrid approach to feature-oriented programming in XVCL* (Zhang and Jarzabek, 2010).

Análise Comparativa dos Vários Mecanismos

Na secção anterior, foram apresentados os vários mecanismos da linguagem de programação C++ que permitem gerir a variabilidade das funcionalidades de um

sistema ao nível do código. A tabela 2.1, apresenta um breve sumário das vantagens e desvantagens de cada uma deles.

Como a maioria dos sistemas de videovigilância não são sistemas adaptativos, isto é, em tempo de execução as configurações do sistema não se alteram de forma dinâmica, é possível em tempo de compilação resolver a variabilidade das funcionalidades do sistema. Assim, no processo de compilação é necessário definir as funcionalidades necessárias para obter a configuração pretendida, para que o compilador gere um ficheiro executável final otimizado para o fim proposto.

Tabela 2.1: Vantagens e desvantagens dos mecanismos de gestão da variabilidade das funcionalidades de um sistema ao nível do código.

Mecanismo	**Vantagens**	**Desvantagens**
Compilação Condicional	- Bem conhecido - Bom desempenho - Alta granularidade	- Difícil compreensão - Difícil manutenção
Polimorfismo Dinâmico	- Bem conhecido - Fácil de utilizar - Dinâmico	- *Overhead* no desempenho - Difícil implementar funcionalidades do tipo OR
Polimorfismo Paramétrico	- Bom desempenho	- Menos conhecido - Menos suportado - *Debugging*
Polimorfismo de Operações	- Bem conhecido	- Menos importante que o polimorfismo universal
C++ *Template Metaprogramming*	- Bom desempenho - Linguagem funcional - Alta granularidade	- Menos conhecido - Menos suportado - Complexidade - *Debugging*
Programação Orientada a Aspetos	- Bom desempenho	- Necessita de ferramentas externas - Aprender nova linguagem - *Debugging*
Frame Technology	Bom desempenho	- Necessita de ferramentas externas - Aprender nova linguagem - *Debugging*

Ao analisar as vantagens e desvantagens dos vários mecanismos de gestão da variabilidade do código da tabela 2.1, conclui-se que o mecanismo C++ *template metaprogramming* é o que mais se adequa para gerir a variabilidade das funcionalidades dos sistemas de videovigilância.

Os fatores que foram tomados em conta para tomar esta decisão foram:

- Mecanismo de gestão da variabilidade estático;

- Permite utilizar técnicas e mecanismos das linguagens funcionais, muito úteis para gerir de forma eficaz a variabilidade de sistemas complexos;
- Não produz *overhead* no desempenho do sistema;
- Apresenta um nível de granularidade semelhante ao apresentado pela compilação condicional;
- Não são necessárias ferramentas externas;
- Não é necessário aprender uma nova linguagem. Isto é, o C++ é utilizado para desenvolver código normal (linguagem imperativa) e o código TMP (linguagem funcional).

2.5 Trabalhos Relacionados

Como a quantidade de trabalhos relacionados com a área de investigação desta tese é considerável, apenas é feita uma breve descrição dos trabalhos que utilizam tecnologias semelhantes ao proposto nesta tese de doutoramento. Essencialmente, são apresentados aqueles que descrevem o desenho e implementação de sistemas de videovigilância e/ou geram a variabilidade das funcionalidades de um sistema.

- Em (Acher et al., 2009), uma mistura de técnicas *model-driven* e *generative programming* são propostas para suportar a variabilidade das configurações dos sistemas de videovigilância. O autor afirma, que para gerir eficazmente a combinação das funcionalidades dos sistemas em tempo de execução, é necessário separar a variabilidade existente ao nível do espaço do domínio, da variabilidade existente ao nível da representação do código. Um dos problemas deste trabalho de investigação é que não é abordada a forma como é gerida a variabilidade das funcionalidades ao nível do código do sistema. Isto é, não apresentado o mecanismo de gestão da variabilidade do código utilizado;
- Em (Lohmann, 2009), é utilizada a programação orientada a aspectos (AOP) para gerir a variabilidade existente no domínio dos sistemas operativos e resolver os problemas relacionados com *crosscutting concerns*. Mesmo sabendo que o mecanismo de gestão da variabilidade AOP, permite um alto nível de granularidade quando comparado outros mecanismos de gestão da variabilidade existentes, pode introduzir *overhead* no desempenho do sistema

dependendo do tipo de *advice* utilizado e da impossibilidade de gerir a variabilidade em sistemas com diversas instâncias da mesma classe, mas com diferentes requisitos de entrada (*code bloat*);

- Em (Sarinho and Apolinário, 2009), os autores propõem o uso de *generative programming* para acelerar o processo de desenvolvimento de jogos digitais, e assim reduzir o *time-to-market*. O objetivo principal da abordagem proposta é integrar os modelos de funcionalidades, as linguagens específicas do domínio (*domain-specific languages*) e os *scripts* de metaprogramação de jogos, de forma a que haja apenas uma única abordagem a seguir na implementação dos jogos. Embora sejam apresentados os diagramas de funcionalidades, as linguagens específicas do domínio e os modelos de transformação do código, restam dúvidas como é implementada a variabilidade ao nível do código dos jogos;

- Em (Buchmann and Dotor, 2009), é apresentado um *plugin* Fujaba que utiliza técnicas *model-driven* para o desenvolvimento de *software product line*. O focos do artigo concentra-se em como fazer a ponte entre a modelação das funcionalidades e o domínio da modelação utilizando anotações;

- Em (AMPLE), é explorado um novo mecanismo por forma a compreender e especificar melhor a relação que existe entre a variabilidade de um sistema e os seus requisitos, combinando SPL, *Aspect-orientation* e técnicas *model-driven*. O projeto *Ample*, apresenta duas linguagens complementares, RDL e VML4RE, que permitem especificar e resolver os problemas de *crosscutting* entre as funcionalidades e os requisitos.

- Em (Wada et al., 2006), é descrita a *framework Matilda*. *Matilda* é uma *framework* de desenvolvimento *model-driven* que permite a geração automática de código através de transformações que utilizam modelos UML, enquanto permite resolver a lacuna na abstração e falha de ferramentas de rastreio relacionadas com as práticas correntes do desenvolvimento *model-driven*;

- Por fim, em (Czarnecki and Eisenecker, 1999), é proposto pelos autores a programação generativa como a forma correta de selecionar de forma automática os componentes e fazer a composição dos mesmos baseando-se no conhecimento das configurações e num conjunto de regras de construção. Um arquitetura em camada, chamada *GenVoca* é utilizada e conhecimento das configurações são implementados utilizando C++ *template metprogramaning*

com gerador.

2.6 Conclusão

Neste capítulo foi apresentado o estado da arte das áreas temáticas subjacentes a esta tese de doutoramento. Numa primeira fase, foi apresentado o processo natural de evolução dos sistemas de videovigilância ao longo dos tempos. De seguida, apresentou-se a arquitetura atual dos sistemas de videovigilância e os problemas associados à forma como estes são desenvolvidos. Como o domínio da videovigilância pode ser visto como uma família de produtos, no passo seguinte foram abordados os conceitos associados ao desenvolvimento de *software* utilizando *Software Product Line*. Assim, foram descritas as várias fases de desenvolvimento do SPL, onde foram enumerados os vários mecanismos de gestão da variabilidade do código e apresentadas as principais vantagens e desvantagens de cada um deles. Por fim, foi feita uma breve apresentação dos trabalhos relacionados.

Assim, conclui-se que: (i) ao desenvolver sistemas complexos, tais como os sistemas de videovigilância, deve-se seguir a abordagem de desenvolvimento de *software* baseado numa família de produtos (*Software Product Line*), em detrimento das tradicionais técnicas de desenvolvimento *ad hoc*, onde o foco de desenvolvimento se concentra apenas num produto específico e (ii) para resolver o problema associado à forma como devem ser implementados os componentes do sistema, deve-se utilizar o *C++ template metaprogramming* como o principal mecanismo de gestão da variabilidade, ao nível da implementação do código.

Capítulo 3

C++ em Ambientes Embebidos

3.1 Introdução

No capítulo anterior foram apresentados os vários mecanismos que permitem gerir a variabilidade do código de um sistema, utilizando a linguagem de programação C++. Entre os mecanismos apresentados, os mais utilizados pela comunidade científica e pela indústria são: a compilação condicional, o polimorfismo dinâmico, o polimorfismo paramétrico e a programação orientada a aspectos (AOP). Cada um dos mecanismos foi classificado em duas categorias: os que mais se adequam para gerir a variabilidade do código de um sistema em tempo de compilação (sistemas estáticos) e os que mais se adequam para gerir a variabilidade do código de um sistema em tempo de execução (sistemas dinâmicos).

Este capítulo apresenta a linguagem de programação C++ como a linguagem de eleição para gerir a variabilidade inerente ao desenvolvimento de aplicações com um grau de complexidade elevado, essencialmente quando estas são compiladas para serem executadas em sistemas embebidos. Assim, numa primeira fase, é feita uma breve introdução da linguagem de programação C++ e os conceitos associados à programação orientada a objetos. Por fim, cada uma das funcionalidades da linguagem de programação C++ é classificada em função dos custos de *overhead* associados à sua implementação em código *assembly* gerado pelo compilador C++ no ato da compilação da aplicação.

3.2 Linguagem de Programação C++

O C++ é uma linguagem de programação multiparadigma e de uso genérico, projetada e desenvolvida por *Bjarne Stroustrup* nos laboratórios Bell, no início dos anos 80. Desde logo, a linguagem de programação C++ tornou-se popular pelo facto de combinar todas as funcionalidades da tradicional linguagem de programação C, com as novas funcionalidades que dão suporte à programação orientada a objetos.

A introdução de novas funcionalidades na especificação do C++ não foi um processo imediato. Ao longo dos tempos, novas funcionalidades foram adicionadas à especificação, tais como, funções virtuais, múltipla herança e tratamento de exceções. Das principais datas que deram origem às novas especificações da linguagem, destacam-se o antigo padrão que foi desenvolvido em 1998 (*C++98*) e que foi atualizado em 2003 (*C++03*). Finalmente em 2011, foi lançada uma nova especificação da linguagem conhecida por *C++11* ou *C++0x*.

Assim, pode-se dizer que a linguagem de programação C++ foi desenvolvida com o intuito de: (i) ser uma linguagem melhor que a tradicional linguagem de programação C, (ii) dar suporte a abstração de dados, (iii) dar suporte ao paradigma de programação orientada a objetos e (iv) suportar programação genérica.

3.2.1 Principais Conceitos da Linguagem

Tal como foi referido anteriormente, a linguagem de programação C++ é uma linguagem com suporte ao paradigma de programação orientada a objeto (OOP). Os principais conceitos de qualquer linguagem OOP são: o encapsulamento, a herança e o polimorfismo.

Encapsulamento

O encapsulamento é um dos conceitos mais importantes de qualquer linguagem OOP. Através do encapsulamento é possível ocultar os detalhes de implementação dos objetos e proteger os dados dos mesmos, de forma a que estes sejam operados apenas pelo próprio objeto sem interferência de entidades externas. Desta forma, o código é dividido em unidades funcionais (classes), que tornam o *software* mais

flexível e fácil de gerir, bem como, o facto de limitar a exposição dos dados de cada classe a qualquer tipo de vulnerabilidade.

O código seguinte define a classe Point2D que implementa um ponto a duas dimensões:

```
class Point2D {
    int _x, _y; // definição das variáveis da classe
public:
    Point2D(int x, int y) // construtor da classe
        : _x(x), _y(y) {}

    void setX(int x) { _x = x; } // definir o valor da variável _x
    int getX() const { return _x; } // ler o valor da variável _x

    void setY(int y) { _y = y; } // definir o valor da variável _y
    int getY() const { return _y; } // ler o valor da variável _y
};
```

A classe Point2D é constituída por duas variáveis de classe: a variável _x e a variável _y. A variável _x define o valor da coordenada do ponto no eixo dos xx e a variável _y define o valor da coordenada do ponto no eixo dos yy. Como, por defeito, os atributos de restrição de acesso de uma classe são privados (*private*), as variáveis da classe são privadas, fazendo com que não seja possível a um objeto de outro tipo aceder de forma direta a cada uma das variáveis.

Para definir e ler os valores de cada uma das variáveis da classe, são definidos os métodos setX, setY, getX e getY. Os métodos são definidos com o atributo de restrição de acesso público (*public*), de forma a que sejam acessíveis a entidades externas à classe.

Herança

O conceito de herança permite que classes já definidas (classes bases) possam ser utilizadas para criar novas classes (*super* classes). Assim sendo, a *super* classe herda todas as propriedades (dados e métodos) das classes bases sem ser necessário defini-las novamente. A herança promove a extensibilidade do código, a sua reutilização e a criação de relações coerentes entre os diferentes objetos.

O código seguinte define a classe Point3D que implementa um ponto a três dimensões:

```
class Point3D : public Point2D {
    int _z; // definição das variáveis da classe
public:
```

```
        Point3D(int x, int y, int z) // construtor da classe
            : Point2D(x, y), _z(z) {}

        void setZ(int z) { _z = z; } // definir o valor da variável _z
        int getZ() const { return _z; } // ler o valor da variável _z
};
```

Como a classe Point3D herda da classe Point2D, todas as propriedades da classe Point2D são herdadas pela classe Point3D. A classe Point3D define uma nova variável de classe com o nome _z, que representa a coordenada do ponto no eixo dos zz. Como a variável é privada, são definidos os métodos públicos setZ e getZ, que permitem definir e ler o valor da variável, respetivamente.

As coordenadas do ponto no eixo do xx e no eixo dos yy são definidas pela classe base Point2D, bem como os métodos públicos que permitem definir e ler os dados das variáveis. No exemplo apresentado, a *super* classe herda apenas de uma classe base, mas caso seja necessário, é possível herdar de várias classes bases (herança múltipla).

Polimorfismo

O conceito de polimorfismo permite atribuir diferentes significados a símbolos ou operadores, em função das suas características ou dos diferentes contextos onde são executados. Na linguagem de programação C++, existem três tipos de polimorfismo: polimorfismo dinâmico (herança/funções virtuais), polimorfismo paramétrico (*templates*) e polimorfismo de operações (*overloading*).

O código seguinte apresenta um exemplo de polimorfismo dinâmico, onde é definida a classe 3D que reimplementa a função virtual definida na classe Point2D:

```
class Point2D {
    ...
public:
    virtual void draw() {
        // desenha o ponto num plano a duas dimensões
    }
};

class Point3D : public Point2D {
    ...
public:
    void draw() {
        // desenha o ponto num plano a três dimensões
    }
};
```

A classe Point3D ao herdar da classe Point2D reimplementa o método draw. O método é reimplementado pela classe Point3D porque a classe base Point2D define o método como virtual e a *super* classe implementa um método com o mesmo nome da classe base. Assim, se o objeto criado for do tipo Point2D, é chamado o método draw da classe Point2D. Caso contrário, se for criado um objeto do tipo Point3D, é chamado o método draw da *super* classe Point3D. Isto apenas funciona se os objetos são criados de forma dinâmica através de variáveis apontadores, tal como ilustra o código seguinte:

```
Point2D *p1 = new Point2D(1, 2);
p1->draw(); // chama o método draw da classe Point2D

Point2D *p2 = new Point3D(1, 2, 3);
p2->draw(); // chama o método draw da classe Point3D
```

3.3 C++ em Sistemas Embebidos

Nos últimos anos, a comunidade que desenvolve *software* para sistemas embebidos, tem debatido muito o facto da utilização da linguagem de programação C++ como a linguagem de eleição para o desenvolvimento de *software* para este tipo de sistemas (Barr, 1998). Embora o *software* para sistemas embebidos seja cada vez mais complexo e a linguagem programação C++ forneça mecanismos que permitem uma melhor gestão do código do que a linguagem de programação C, esta última continua a ser a linguagem de eleição por parte da comunidade, no que diz respeito ao desenvolvimento de *software* para sistemas embebidos.

Em (Barr, 1998), o autor refere que são muitos os mitos que inibem a comunidade de utilizar a linguagem de programação C++ em sistemas embebidos. Segundo o autor, alguns dos argumentos que fazem com que o C++ não seja utilizado em sistemas embebidos são: (i) o C++ não oferece desempenho adequado, (ii) o C++ produz executáveis grandes, (iii) a abstração leva à ineficiência, (iv) os objetos são grandes e (v) as funções virtuais são lentas. A grande razão para isto acontecer, é que os programadores sabem sempre muito mais acerca da linguagem de programação C do que da linguagem de programação C++. Logo, para que estes mitos desapareçam das suas mentes, os programadores têm que perceber como realmente as funcionalidades da linguagem de programação C++ são implementadas em código *assembly*.

Em (Herity), o autor defende que se deve utilizar o *Embedded* C++ no desen-

volvimento de *software* para sistemas embebidos. No artigo, o autor defende que determinadas funcionalidades da linguagem de programação C++ produzem um impacto negativo no desempenho e no tamanho do ficheiro executável das aplicações, especialmente quando estas são executadas em sistemas embebidos. Como o *Embedded* C++ é um subconjunto da linguagem de programação C++ onde foram removidas as funcionalidades do C++ que provocam *overhead* nas aplicações, o *Embedded* C++ pode ser considerada a linguagem ideal para desenvolver *software* para sistemas embebidos.

Para se identificar as funcionalidades da linguagem C++ que provocam *overhead* nas aplicações que são compiladas para serem executadas em sistemas embebidos, a próxima secção explica a forma como cada uma das funcionalidades da linguagem de programação C++ são implementadas em linguagem de programação C. Para verificar o impacto que as funcionalidades da linguagem de programação C++ provoca no código binário da aplicação, é apresentado o código *assembly* gerado pelo compilador Microsoft Visual C++ ao compilar cada uma das funcionalidades.

3.3.1 Classes

Em C++, o conceito de encapsulamento é implementado através de classes. A classe adiciona à estrutura em C, a possibilidade de definir funções membros e mecanismos de ocultação de informação (*public*, *protected* e *private*). Aliás, em C++ o único fator diferenciador entre uma classe e uma estrutura, é que por defeito os dados da classe são privados e os dados da estrutura são públicos.

O código seguinte apresenta a definição da classe Point2D que implementa um ponto a duas dimensões:

```
class Point2D {
        int _x;
        int _y;
public:
        void init(int x, int y) {
                _x = x;
                _y = y;
        }
};
```

A classe Point2D é constituída por duas variáveis membros: a variável _x e a variável _y. A variável _x representa o valor da coordenada do ponto no eixo dos xx e a variável _y representa o valor da coordenada do ponto no eixo dos yy. Como as

variáveis membros são declaradas como privadas, não é possível aceder a cada uma delas do exterior. Para que seja possível definir o valor das variáveis membros do exterior, a função membro init é declarada como pública. Assim, a função membro init permite definir os valores das coordenadas do ponto com os valores dos parâmetros de entrada x e y.

A figura 3.1 apresenta a forma como os dados da classe Point2D estão organizados em memória:

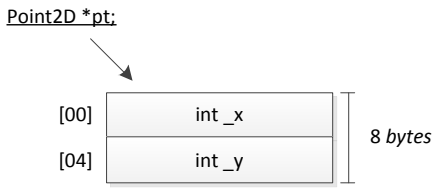


Figura 3.1: Disposição dos dados da classe Point2D em memória.

No total, os dados da classe Point2D ocupam 8 *bytes* em memória. Os primeiros 4 *bytes* são reservados para a variável membro _x e os 4 *bytes* seguintes são reservados para a variável membro _y.

Implementação em C

O código seguinte apresenta a definição da estrutura de dados Point2D que implementa um ponto a duas dimensões:

```
struct Point2D {
        int _x;
        int _y;
};
```

Tal como na definição da classe Point2D, a estrutura de dados Point2D é constituída pela variável _x e pela variável _y, que representam as coordenadas do ponto no eixo dos xx e no eixo dos yy, respetivamente. Embora os dados da classe e da estrutura ocupem o mesmo espaço em memória, os dados da classe são definidos por defeito como privados e os dados da estrutura são definidos por defeito como públicos, logo podem ser acedidos do exterior.

Como as estruturas em C não suportam o conceito de funções membros, é criada a função Point2D_Init que permite inicializar o ponto apontado pelo apontador this, com os valores dos parâmetros de entrada x e y:

```
void Point2D_Init(struct Point2D *this, int x, int y) {
    this->_x = x;
    this->_y = y;
}
```

Ao contrário da função membro init da classe Point2D que recebe apenas dois parâmetros de entrada, a função Point2D_Init recebe três parâmetros de entrada. Isso deve-se ao facto de na função em C ser necessário passar como parâmetro de entrada o apontador this, que aponta para os dados da estrutura que representa o ponto a definir. Em C++, o apontador this da classe é passado de forma automática pelo compilador à função membro init da classe Point2D.

Ao analisar o código das duas implementações, verifica-se que a implementação em C++ utilizando classe é muito mais percetível e segura que a mesma implementação em C utilizando estruturas de dados. Isso deve-se ao facto de ser possível numa classe definir funções membros e mecanismos de ocultação de informação, fazendo com que o código seja mais modular e protegido.

Código *Assembly*

Ao compilar o código C++ da classe Point2D, o compilador C++ gera o seguinte código *assembly* para a função membro init:

```
?init@Point2D@@QAEXHH@Z proc near
    ...
    mov eax, ecx ; o registo eax toma o valor do apontador da classe this
    mov edx, [ebp+8] ; o registo edx toma o valor do parâmetro de entrada x
    mov [eax], edx ; a variável membro _x toma o valor de x
    mov edx, ecx ; o registo edx toma o valor do apontador da classe this
    mov eax, [ebp+0Ch] ; o registo eax toma o valor do parâmetro de entrada y
    mov [edx+4], eax ; a variável membro _y toma o valor de y
    ...
?init@Point2D@@QAEXHH@Z endp
```

O código *assembly* gerado pelo compilador C++ é muito similar ao código *assembly* gerado pelo compilador C ao compilar a versão C utilizando estruturas de dados. A única diferença digna de registo, reside no facto de na implementação em C++ o compilador utilizar o registo ecx para apontar para os dados da classe (apontador this da classe) e na implementação em C o apontador this ser passado com parâmetro de entrada da função Point2D_Init através da pilha.

Assim, conclui-se que as classes são processadas pelo *front-end* do compilador C++, não sendo produzido qualquer tipo de custo ao nível do desempenho e do

tamanho do código executável de uma aplicação.

3.3.2 *Namespace*

O conceito de *namespace* permite agrupar entidades em C++, tais como, classes, objetos e funções, através de um nome. Desta forma, é possível criar diferentes entidades com o mesmo nome sem causar erros de compilação.

O código seguinte define o *namespace* Plano2D e o *namespace* Plano3D:

```
namespace Plano2D {
    int _x0;
};

namespace Plano3D {
    int _x0;
};
```

O *namespace* Plano2D define todas as figuras geométricas possíveis de um plano a duas dimensões. Para definir todas as figuras geométricas de um plano a três dimensões é definido o *namespace* Plano3D. Por forma a simplificar a implementação de cada um dos planos, é dado apenas o exemplo da definição da variável _x0, que define a coordenada inicial do plano no eixo dos xx.

O código seguinte inicializa a variável _x0 do *namespace* Plano2D a 1 e a variável _x0 do *namespace* Plano3D a 5:

```
Plano2D::_x0 = 1;
Plano3D::_x0 = 5;
```

Implementação em C

O código seguinte define a variável plano2d_x0 do plano a duas dimensões e a variável plano3d_x0 do do plano a três dimensões:

```
#ifdef _PLANO2D_
    int plano2d_x0;
#endif

#ifdef _PLANO3D_
    int plano3d_x0;
#endif
```

Como não existe nenhuma funcionalidade em C que implemente os *namespaces* na sua totalidade, é utilizada a compilação condicional. Para que não haja conflitos

de nomes entre as variáveis dos dois planos, o nome de cada uma delas começa com o nome do *namespace* onde estas são definidas em C++. Assim, são definidas as variáveis de compilação _PLANO2D_ e _PLANO3D_ para definir os *namespaces* em C, e as variáveis plano2d_x0 e plano3d_x0 para definir as coordenadas iniciais de cada um dos planos no eixo dos xx.

O código seguinte inicializa a variável plano2d_x0 do plano a duas dimensões a 1 e a variável plano3d_x0 do plano a três dimensões a 5:

```
#ifdef _PLANO2D_
    plano2d_x0 = 1;
#endif
#ifdef _PLANO3D_
    plano3d_x0 = 5;
#endif
```

Ao analisar o código das duas implementações, verifica-se que a implementação em C++ utilizando *namespaces* é muito mais fácil de implementar e percetível que a mesma implementação em C utilizando compilação condicional e diferentes nomes para as variáveis de cada um dos planos. Ao utilizar *namespace* é possível atribuir nomes iguais a variáveis de diferentes *namespaces*, sendo da responsabilidade do compilador C++ a atribuição automática de nomes diferentes para as variáveis através da técnica designada por *name mangling*.

Código *Assembly*

Ao compilar o código C++ que inicializa as variáveis _x0 dos dois *namespaces* a 1 e 5, o compilador C++ gera o seguinte código *assembly*:

```
...
mov ?x@Plano2D@@3HA, 1 ; inicializa a variável x do namespace Plano2D a 1
mov ?x@Plano3D@@3HA, 5 ; inicializa a variável x do namespace Plano3D a 5
...
```

O código *assembly* gerado pelo compilador C++ é muito similar ao código gerado pelo compilador C ao compilar a versão C utilizando compilação condicional e nomes diferentes para as variáveis de cada um dos planos. A única diferença digna de registo, reside no facto de na implementação em C++ o compilador gerar de forma automática nomes diferentes para as duas variáveis através da técnica *name mangling*, em que no caso da versão em C tem que ser o programador a atribuir nomes diferentes para as variáveis dos diferentes planos.

Logo, como os *namespaces* são processados pelo *front-end* do compilador C++, conclui-se que a utilização de *namespaces* não produz qualquer tipo de custo ao nível do desempenho e do tamanho do código executável de uma aplicação.

3.3.3 Funções *Inline*

As funções *inline* são uma forma mais legível e segura de implementar macros em C, especialmente quando estas são utilizadas em funções ordinárias. Como *inline* é apenas uma diretiva ao compilador, significa que o compilador em determinados casos pode ignorar a substituição do código da chamada da função pelo código do corpo da função. Isto é, por defeito os compiladores definem um número máximo de funções *inline* que são expandidas. A partir desse número, todas as funções *inline* que apareçam não são expandidas, fazendo com que no ato da compilação sejam apresentadas *warnings* a informar do sucedido. Para que as funções *inline* sejam sempre expandidas, alguns compiladores, tal como o *gcc*, permitem definir as funções *inline* com o atributo *always*, que neste caso faz com que o *inline* deixe de ser uma diretiva mas sim uma ordem.

O código seguinte define a função membro init da classe Point2D como *inline*:

```
inline void Point2D::init(int x, int y) {
    _x = x;
    _y = y;
}
```

Para definir a função membro init da classe Point2D como *inline*, é necessário introduzir a palavra chave do C++ *inline* antes do tipo de dados de retorno da função, ou tal como no exemplo anterior, implementar a função membro init dentro da definição da classe Point2D.

O código seguinte cria uma instância da classe Point2D e chama a função membro init da classe Point2D com os parâmetros de entrada x e y a 0:

```
Point2D pt;
pt.init(0, 0);
```

Implementação em C

Para implementar a mesma funcionalidade da função *inline* init da classe Point2D em C, é necessário definir a macro INIT_POINT, que inicializa o ponto pt com os valores das variáveis x e y:

```
#define INIT_POINT(pt, x, y)\
        pt._x = x;\
        pt._y = y;
```

Para criar uma instância da estrutura de dados Point2D e chamar a macro INIT_POINT com os valores dos parâmetros de entrada x e y a 0, são necessárias as seguintes linhas de código:

```
struct Point2D pt;
INIT_POINT(pt, 0, 0);
```

Ao analisar o código das duas implementações, verifica-se que a implementação em C++ utilizando funções *inline* é muito mais segura que o código da implementação em C utilizando macros. O código é mais seguro utilizando funções *inline* porque a função só pode ser chamada num objeto do tipo Poin2D e os parâmetros de entrada da função têm de ser do tipo inteiro. No caso das macros em C, não é possível especificar o tipo de dados de entrada e nem o *scope* da macro.

Código *Assembly*

Ao compilar o código C++ da chamada da função inline init da classe Point2D, o compilador C++ gera o seguinte código *assembly*:

```
...
xor ecx, ecx ; o registo ecx toma o valor de 0
...
mov [esi], ecx ; inicializa o valor da variável membro x do ponto pt a 0
mov [esi+4], ecx; inicializa o valor da variável membro y do ponto pt a 0
...
```

O código *assembly* gerado pelo compilador C++ é igual ao código gerado pelo compilador C ao compilar a versão C utilizando macros. Tal como acontece com as macros, ao definir uma função como *inline* é necessário ter em consideração o número de linhas de código da função. Se o número de linhas de código da função for considerável e a função for chamada muitas vezes no código fonte da aplicação, a substituição da chamada da função pelo corpo da mesma faz com que o tamanho do ficheiro executável da aplicação aumente de forma significativa.

Logo, conclui-se que a utilização devida de funções *inline*, não produz qualquer tipo de custo ao nível do desempenho e do tamanho do código executável de uma aplicação.

3.3.4 *Overloading* de Operadores

O *overloading* de operadores permite utilizar operadores *standard* para realizar operações com classes, tal como se utilizam nos tipos fundamentais da linguagem de programação C++.

O código seguinte define o operador += da classe Point2D:

```
const Point2D& Point2D::operator+=(const Point2D& pt) {
    _x += pt._x;
    _y += pt._y;
    return *this;
}
```

O operador += da classe Point permite adicionar às coordenadas do ponto em questão, as coordenadas do ponto pt, que é passado como parâmetro de entrada à função.

O código seguinte cria a duas instâncias de um ponto a duas dimensões, p1 e p2, e chama o operador += do ponto p1. O ponto p2 é passado como parâmetro de entrada, permitindo que seja adicionado às coordenadas do ponto p1, as coordenadas do ponto p2:

```
Point2D p1, p2;
p1 += p2;
```

Implementação em C

Para implementar a mesma funcionalidade em linguagem C, são necessárias as seguintes linhas de código:

```
void Point2D_Add(struct Point2D *p1, struct Point2D *p2) {
    p1->_x = p1->_x + p2->_x;
    p1->_y = p1->_y + p2->_y;
}
```

Como a linguagem C não suporta o *overloading* de operadores, é definida a função Point2D_Add que adiciona ao ponto p1, o ponto p2. O ponto p1 e o ponto p2 são recebidos como parâmetros de entrada da função e são do tipo apontador para a estrutura de dados Point2D.

O código seguinte cria o ponto p1 e o ponto p2, e chama a função Point2D_Add para adicionar ao ponto p1, o ponto p2:

```
struct Point2D p1, p2;
Point2D_Add(&p1, &p2);
```

Ao analisar o código das duas implementações, verifica-se que a implementação em C++ utilizando *overloading* de operadores é muito mais percetível que a mesma implementação em C utilizando funções.

Código *Assembly*

Ao compilar o código C++ do operador += da classe Point2D, o compilador C++ gera o seguinte código *assembly*:

```
??YPoint2D@@QAEABV0@ABV0@@Z proc near
    ...
    mov eax, [ebp+this] ; o registo eax toma o valor do apontador this da classe
    mov ecx, [eax] ; o registo ecx toma o valor da variável membro _x da classe
    mov edx, [ebp+pt] ; o registo edx toma o valor da variável membro _x do ponto pt
    add ecx, [edx] ; o registo ecx toma o valor da soma das variáveis membros _x
                   ; da classe e do ponto pt
    mov eax, [ebp+this] ; o registo eax toma o valor do apontador this da classe
    mov [eax], ecx ; a variável membro _x da classe toma o valor da soma
    mov ecx, [ebp+this] ; o registo ecx toma o valor do apontador this da classe
    mov edx, [ecx+4] ; o registo edx toma o valor da variável membro _y da classe
    mov eax, [ebp+pt] ; o registo eax toma o valor do endereço de memória do ponto pt
    add edx, [eax+4] ; o registo edx toma o valor da soma das variáveis membros _y
                     ; da classe e do ponto pt
    mov [ecx+4], edx ; a variável membro _y da classe toma o valor da soma
    ...
??YPoint2D@@QAEABV0@ABV0@@Z endp
```

O código *assembly* gerado pelo compilador C++ é similar ao código gerado pelo compilador C ao compilar a versão C utilizando funções. A única diferença digna de registo é que o compilador C++ gera de forma automática o nome da função que representa o operador += através da técnica *name mangling*.

Como o *overloading* de operadores é processado pelo *front-end* do compilador C++, conclui-se que a utilização de *overloading* de operadores não produz qualquer tipo de custo ao nível do desempenho e do tamanho do código executável de uma aplicação.

3.3.5 Construtor e Destrutor

O construtor é uma função membro que é executada sempre que um objeto é criado. Isto significa, que sempre que um objeto é instanciado, o compilador reserva espaço em memória para o objeto e chama de forma automática o construtor para inicializar os dados do objeto. De forma similar, o destrutor é chamado sempre

que um objeto fica fora do âmbito da execução, para libertar os recursos alocado pelo objeto.

O código seguinte implementa o construtor da classe Point2D:

```
class Point2D {
    ...
public:
    Point2D(int x, int y)
        : _x(x), _y(y)
    {}
    ...
};
```

O construtor da classe Point2D inicializa a variável membro _x e a variável membro _y a zero. O destrutor da classe não é apresentado porque funciona de forma análoga ao construtor, só que em vez de ser chamado sempre que uma instância do objeto é criada é chamado antes de se libertar a memória do objeto.

O código seguinte cria um ponto a duas dimensões com a coordenada do eixo dos xx a 1 e e a coordenada do eixo dos yy a 2:

```
Point2D point2D(1, 2);
```

Implementação em C

Para implementar a mesma funcionalidade em linguagem de programação C, são necessárias as seguintes linhas de código:

```
void Point2D_Constructor(struct Point2D* pt, int x, int y) {
    pt->_x = x;
    pt->_y = y;
}
```

A função Point2D_Constructor recebe como parâmetros de entrada: um apontador do tipo Point2D que aponta para o ponto a inicializar e os valores das variáveis x e y para inicializar as coordenadas do ponto. Sempre que é reservado espaço em memória para a estrutura de dados Point2D da variável pt, é necessário chamar de seguida a função Point2D_Constructor para inicializar o ponto.

O código seguinte mostra um exemplo onde é criada uma instância de um ponto a duas dimensões e é chamado a função que inicializa o ponto com os valores das coordenadas do ponto na origem:

```
struct Point2D pt;
Point2D_Constructor(&pt, 0, 0);
```

Ao analisar o código das duas implementações, verifica-se que a implementação em C++ utilizando construtores/destrutores de classes é muito mais percetível e intuitiva que a mesma implementação em C utilizando funções. Quando é criado um objeto em C++, o compilador C++ chama de forma automática o construtor da classe. No caso da implementação em C, é da responsabilidade do programador chamar a função que inicializa o ponto.

Código *Assembly*

Ao compilar o código C++ do construtor da classe Point2D, o compilador C++ gera o seguinte código *assembly*:

```
??0Point2D@@QAE@XZ proc near
        ...
        mov eax, [ebp+this] ; o registo eax toma o valor do apontador this da classe
        mov dword ptr [eax], 0 ; a variável membro _x da classe toma o valor de 0
        mov ecx, [ebp+this] ; o registo ecx toma o valor do apontador this da classe
        mov dword ptr [ecx+4], 0 ; a variável membro _y da classe toma o valor de 0
        ...
??0Point2D@@QAE@XZ endp
```

O código *assembly* gerado pelo compilador C++ é similar ao código gerado pelo compilador C ao compilar a versão C utilizando funções.

Como o construtor/destrutor de classe é processado pelo *front-end* do compilador C++, conclui-se que a utilização de construtores/destrutores de classes não produz qualquer tipo de custo ao nível do desempenho e do tamanho do código executável de uma aplicação.

3.3.6 Referências

As referências são semelhantes a um apontador em C. O único fator diferenciador entre os dois é que como as referências tem menos poder de acesso aos dados, permitem aceder aos dados de uma forma mais segurança.

O código seguinte apresenta um exemplo da utilização de referências:

```
class Point2D {
        ...
public:
        Point2D(const Point2D& pt)
           : _x(pt._x), _y(pt._y)
        {}
        ...
```

```
};
```

O *copy* construtor da classe Point2D permite que o valor das coordenadas do ponto possam ser inicializadas através das coordenadas de outro ponto. O *copy* construtor recebe como parâmetro de entrada uma referência para um objeto do tipo Point2D, e inicializa o ponto com os mesmo valores das variáveis _x e _y que o ponto pt.

O código seguinte cria o ponto p1 e inicializa as coordenadas do ponto p2 com os valores das coordenadas do ponto p1:

```
Point2D p1(1, 2);
Point2D p2(p1);
```

Implementação em C

Para implementar a mesma funcionalidade em linguagem C, são necessárias as seguintes linhas de código:

```
void Point2D_Copy_Constructor(struct Point2D* p1, struct Point2D* p2) {
    p1->_x = p2->_x;
    p1->_y = p2->_y;
}
```

Em linguagem C, uma forma de implementar a passagem por referência é utilizando apontadores. Assim, o método Point2D_Copy_Constructor recebe como parâmetros de entrada: um apontador para o ponto a inicializar (p1) e um apontador para o ponto a copiar (p2).

O código seguinte cria o ponto p1 e o ponto p2 e inicializa o ponto p1 com os valores 1 e 2. Para inicializar as coordenadas do ponto p2 com as coordenadas do ponto p1 é chamada a função Point2D_Copy_Constructor:

```
struct Point2D p1, p2;
Point2D_Constructor(&p1, 1, 2);
Point2D_Copy_Constructor(&p2, &p1);
```

Ao analisar o código das duas implementações, verifica-se que a implementação em C++ utilizando referências é mais percetível e seguro que a mesma implementação em C utilizando apontadores.

Código *Assembly*

Ao compilar o código C++ do *copy* construtor da classe Point2D, o compilador C++ gera o seguinte código *assembly*:

```
??0Point2D@@QAE@ABV0@@Z proc near
        ...
        mov ecx, [ebp+pt] ; o registo ecx toma o valor do parâmetro de entrada
                          ; com a referência para o ponto a copiar
        ...
??0Point2D@@QAE@ABV0@@Z endp
```

Ao analisar o código *assembly* das duas implementações, conclui-se que o código gerado pelo compilador C++ utilizando referências é igual ao código gerado pelo compilador C utilizando apontadores.

Como as referências são processadas pelo *front-end* do compilador C++, conclui-se que a utilização de referências não produz qualquer tipo de custo ao nível do desempenho e do tamanho do ficheiro executável de uma aplicação.

3.3.7 Herança Única

O conceito de herança única permite que uma *super* classe herde apenas de uma única classe base. O exemplo seguinte apresenta um caso de herança única, onde a *super* classe Point3D herda da classe base Point2D:

```
class Point3D : public Point2D {
        int _z;
public:
        Point3D(int x, int y, int z)
           : Point2D(x, y), _z(z)
        {}
};
```

A *super* classe Point3D implementa um ponto a três dimensões, onde a base de implementação da *super* classe é a classe base Point2D. Assim, a *super* classe Point3D herda todas as propriedades da classe base Point2D e necessita apenas de definir a variável _z que representa a componente do eixo dos zz.

A figura 3.2 apresenta a forma como os dados da classe Point3D estão organizados em memória:

A classe Point3D ocupa 12 *bytes*: os primeiros 8 *bytes* são reservados para a classe Point2D e os restantes 4 *bytes* para a variável membro _z. Como os dados da classe

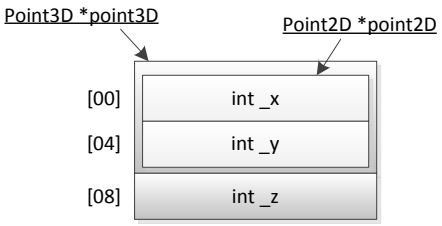


Figura 3.2: Disposição dos dados da classe Point3D em memória.

Point2D ocupam os primeiros *bytes* da class Point3D é possível fazer a conversão de dados da classe Point3D para a classe Point2D de forma direta, sem ser necessário adicionar um offset ao endereço de memória dos dados da classe Point3D.

O código seguinte apresenta um exemplo da conversão de dados da classe Point3D para a classe Point2D:

```
Point3D point3D(1, 2, 3);
Point2D *point2D = &point3D;
```

Implementação em C

Para implementar a mesma funcionalidade do exemplo anterior em C, é necessário em primeiro lugar definir a estrutura de dados Point3D:

```
struct Point3D {
        struct Point2D pt;
        int _z;
};
```

A estrutura de dados Point3D é constituída pela variável pt que representa um ponto a duas dimensões e pela variável _z que define a componente do ponto no eixo dos zz. O código seguinte apresenta a definição da função que inicializa os dados do ponto a três dimensões:

```
void Point3D_Constructor(struct Point3D* pt,
                int x, int y, int z) {
        Point2D_Constructor((struct Point2D*)pt, x, y);
        this->_z = z;
}
```

A função que inicializa as coordenadas do ponto pt a três dimensões com os valores x, y e z, chama numa primeira fase a função que inicializa os dados do ponto a duas dimensões e por fim inicializa a componente do ponto do eixo dos zz com o valor de z.

Para criar uma variável do tipo Point3D e obter um apontador para a estrutura de dados do tipo Point2D do ponto, são necessárias as seguintes linhas de código:

```
struct Point3D point_3d;
Point3D_Constructor(point_3d, 1, 2, 3);
struct Point2D *point_2d = ((struct Point2D*)&point_3d);
```

Ao analisar o código das duas implementações, verifica-se que a implementação em C++ utilizando herança única é mais fácil de implementar, ocupa menos linhas de código e percetível que a mesma implementação em C utilizando estruturas de dados.

Código *Assembly*

Para verificar se a herança única produz algum custo ao nível do desempenho e do tamanho do ficheiro executável, é apresentado o código *assembly* do construtor da classe Point3D gerado pelo compilador C++:

```
??0Point3D@@QAE@HHH@Z proc near
    ...
    mov [ebp+this], ecx ; inicializa o valor da variável da classe _z a zero
    mov eax, [ebp+y] ; coloca no registo eax o valor da variável y
    push eax ; coloca o valor da variável y na pilha (2º parâmetro)
    mov ecx, [ebp+x] ; coloca no registo ecx o valor da variável x
    push ecx ; coloca o valor da variável x na pilha (1º parâmetro)
    mov ecx, [ebp+this] ; coloca no registo ecx o valor do apontador this da classe
    call ??0Point2D@@QAE@HH@Z ; chama o construtor da classe base Point2D
    ...
??0Point3D@@QAE@HHH@Z endp
```

Ao analisar o código assembly do construtor da classe Point3D, verifica-se que o código é muito similar ao código gerado pelo compilador C ao compilar a versão C.

Assim, conclui-se que a utilização de herança única não apresenta qualquer tipo de custo no desempenho ou no aumento do tamanho do ficheiro executável da aplicação.

3.3.8 Herança Múltipla

O conceito de herança múltipla permite que uma *super* classe herde de várias classes bases. O exemplo seguinte apresenta um caso de herança múltipla, onde é definida a *super* classe SharedPoint3D:

```
class SharedPoint3D : public Point3D, public Shared {
        Mutex _mutex;
public:
        SharedPoint3D(int x, int y, int z)
            : Point3D(x, y, z), Shared(&_mutex)
        {}
};
```

A *super* classe SharedPoint3D implementa um ponto a três dimensões que pode ser partilhado por várias *threads*. Para implementar as funcionalidades de um ponto a três dimensões, a *super* classe SharedPoint3D herda da classe base Point3D e para implementar as funcionalidades de sincronismo entre *threads*, a *super* classe SharedPoint3D herda da classe base Shared. A classe SharedPoint3D define a variável membro _mutex que é uma classe do tipo Mutex, que é utilizada pela classe base Shared para fazer o sincronismo entre *threads*.

A figura 3.3 apresenta a forma como os dados da classe SharedPoint3D estão organizados em memória:

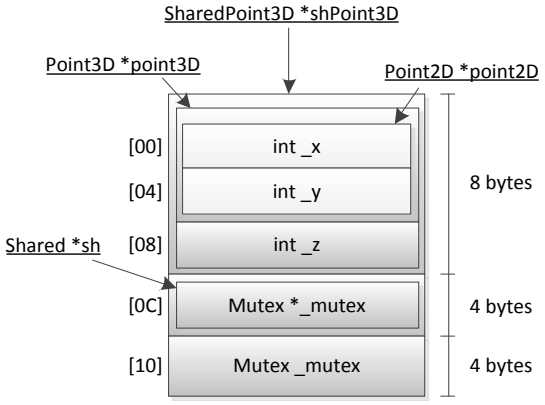


Figura 3.3: Disposição dos dados da classe SharedPoint3D em memória.

No total, os dados da *super* classe SharedPoint3D ocupam 16 *bytes*. Os primeiros 8 *bytes* são reservados para a classe base Point3D. Os 4 *bytes* seguintes são reservados para a classe base Shared. Por fim, os 4 *bytes* restantes são reservados para a variável membro _mutex da classe.

O código seguinte permite obter uma instância de cada uma das classes bases que a *super* classe SharedPoint3D herda:

```
SharedPoint3D sharedPoint3D(1, 2, 3);
Point3D *point3D = &sharedPoint3D;
Shared *shared = &sharedPoint3D;
```

A variável apontador point3D aponta para a instância do ponto sharedPoint3D do tipo Point3D. A variável apontador shared aponta para a instância do ponto sharedPoint3D do tipo Shared. Como a classe Point3D ocupa os primeiros 8 *bytes* da memória de dados da classe SharedPoint3D, a conversão de dados da classe SharedPoint3D para a classe Point3D é direta. Por sua vez, a conversão de dados da classe SharedPoint3D para a classe Shared é indireta. Isto é, antes de se obter o apontador para os dados da classe Shared da classe SharedPoint3D, é necessário adicionar 12 *bytes* ao endereço de memória da classe SharedPoint3D.

Implementação em C

Para implementar a mesma funcionalidade do exemplo anterior em C, é necessário definir a estrutura de dados SharedPoint3D:

```
struct SharedPoint3D {
        struct Point3D pt;
        struct Shared sh;
        struct Mutex _mutex;
};
```

A estrutura de dados SharedPoint3D é constituída por três variáveis: a estrutura de dados pt do tipo Point3D, a estrutura de dados sh do tipo Shared e a estrutura de dados _mutex do tipo Mutex. Para a estrutura de dados SharedPoint3D implementar a mesma funcionalidade que a classe SharedPoint3D, as variáveis da estrutura SharedPoint3D têm que seguir a mesma ordem de definição das classes bases da classe SharedPoint3D.

O código seguinte apresenta a função SharedPoint3D_Constructor que inicializa um ponto a três dimensões partilhado:

```
void SharedPoint3D_Constructor(struct SharedPoint3D* this, int x, int y, int z) {
            Point3D_Constructor((struct Point3D*)&this->pt, x, y, z);
            Shared_Constructor((struct Shared*)&this->sh, &this->_mutex);
}
```

A função SharedPoint3D_Constructor recebe três parâmetros de entrada: o ponto a três dimensões a inicializar (apontador this) e o valor das coordenadas dos três eixos a definir no ponto (x, y e z). A função chama numa primeira fase a função Point3D_Constructor que inicializa as componentes do ponto a três dimensões. Por fim, chama a função Shared_Constructor para inicializar o objeto de sincronização sh da classe SharedPoint3D.

Para criar uma estrutura de dados do tipo SharedPoint3D e obter um apontador para as estrutura de dados do tipo Point3D e um apontador para as estrutura de dados

do tipo Shared do ponto, são necessárias as seguintes linhas de código:

```
struct SharedPoint3D sh_point_3d;
SharedPoint3D_Constructor(sh_point_3d, 1, 2, 3);
struct Point3D *point_3d = ((struct Point3D*)&sh_point_3d);
struct Shared *shared = ((struct Shared*)&sh_point_3d);
```

Ao analisar o código das duas implementações, verifica-se que a implementação em C++ utilizando herança mútipla é muito mais fácil de implementar, ocupa menos linhas de código e muito mais percetível que a mesma implementação em C utilizando estruturas de dados.

Código *Assembly*

Para verificar se a herança múltipla produz algum custo ao nível do desempenho e do tamanho do ficheiro executável, é apresentado o código *assembly* do construtor da classe SharedPoint3D gerado pelo compilador C++:

```
??0SharedPoint3D@@QAE@HHH@Z proc near
    ...
    mov eax, [ebp+z] ; o registo eax toma o valor do parâmetro de entrada z
    push eax ; insere na pilha o valor de z
    mov ecx, [ebp+y] ; o registo ecx toma o valor do parâmetro de entrada y
    push ecx ; insere na pilha o valor de y
    mov edx, [ebp+x] ; o registo edx toma o valor do parâmetro de entrada x
    push edx ; insere na pilha o valor de x
    mov ecx, [ebp+this] ; o registo ecx toma o valor do apontador this da classe
    call ??0Point3D@@QAE@HHH@Z ; chama o construtor Point3D::Point3D(x, y, z)
    ...
    push eax ; o registo eax toma o valor do endereço de memória da variável da classe _mutex
    mov ecx, [ebp+this] ; o registo ecx toma o valor do apontador this da classe
    call ??0Shared@@QAE@PAVMutex@@@Z ; chama o construtor Shared::Shared(&_mutex)
    ...
??0SharedPoint3D@@QAE@HHH@Z endp
```

Tal como na função SharedPoint3D_Constructor da implementação em C, o construtor da classe SharedPoint3D chama numa primeira fase a função ??0Point3D@@QAE@HHH@Z e depois a função ??0Shared@@QAE@PAVMutex@@@Z. A função ??0Point3D@@QAE@HHH@Z é o construtor da classe Point3D e a função ??0Shared@@QAE@PAVMutex@@@Z é construtor da classe Shared.

O código *assembly* seguinte permite obter uma instância de cada uma das classes bases que a *super* classe SharedPoint3D herda:

```
...
sub esp, 20h ; reserva espaço na pilha para as variáveis locais
push 3 ; coloca na pilha o parâmetro de entrada z
push 2 ; coloca na pilha o parâmetro de entrada y
```

```
push 1 ; coloca na pilha o parâmetro de entrada x
lea ecx, [ebp+sharedPoint3D] ; o registo ecx toma o valor de memória do objeto sharedPoint3D
call ??0SharedPoint3D@@QAE@HHH@Z ; chama o construtor da classe SharedPoint3D
lea eax, [ebp+sharedPoint3D] ; o registo eax toma o valor de memória do objeto sharedPoint3D
mov [ebp+point3D], eax ; a variável shared toma o valor do endereço de memória da conversão
                                    ; da classe SharedPoint3D para a classe Point3D
...
lea edx, [ebp+sharedPoint3D] ; o registo edx toma o valor de memória do objeto sharedPoint3D
add edx, 0Ch ; conversão da classe SharedPoint3D para a classe Shared
mov [ebp+shared], edx ; a variável shared toma o valor do endereço de memória da conversão
...
```

A primeira linha de código reserva espaço em memória na pilha para as três variáveis locais: a variável sharedPoint3D que é do tipo SharedPoint3D, a variável point3D que é um apontador do tipo Point3D e a variável shared que é um apontador do tipo Shared.

Para inicializar o objeto sharedPoint3D, é chamado o construtor da classe SharedPoint3D com o parâmetro de entrada 1 (inicializa a coordenada do ponto no eixo dos xx), o parâmetro de entrada 2 (inicializa a coordenada do ponto no eixo dos yy) e o parâmetro de entrada 3 (inicializa a coordenada do ponto no eixo dos zz).

Depois de inicializado o objeto sharedPoint3D, é obtida a instância do ponto do tipo Point3D através da variável point3D. A conversão de dados é direta porque a classe Point3D é a primeira classe base donde a *super* classe SharedPoint3D herda, o que implica que os dois tipos de dados ocupem o mesmo endereço de memória.

Por fim, é obtida a instância do ponto do tipo Shared através da variável shared. Para fazer a conversão entre os dois tipos de dados, é necessário adicionar ao endereço de memória da *super* classe SharedPoint3D, 12 *bytes* (o número de *bytes* que ocupa um objeto do tipo Point3D).

Assim, conclui-se que para que um apontador de uma classe base tente aceder aos dados de uma *super* classe, e a classe base não é a primeira da lista de heranças, é necessário adicionar um *offset* ao endereço de memória da *super* classe. O que implica que existe uma indirecção no acesso aos dados da classe base. Logo, a utilização de herança múltipla produz custos ao nível do desempenho e do tamanho do ficheiro executável de uma aplicação.

3.3.9 Funções Virtuais

As funções virtuais implementam um dos conceitos fundamentais das linguagens orientadas a objeto, o polimorfismo. Basicamente, as funções virtuais permitem implementar o principio da substituição ao utilizar herança. Isto é, se uma classe A herda de uma classe B e reimplementa as funções virtuais da classe A, é possível chamar as novas funções implementadas pela classe B, mesmo sabendo apenas as informações em tempo de execução de uma classe do tipo A.

O código seguinte define a função virtual draw da classe Point2D:

```
class Point2D {
        int _x, _y;
public:
        ...
        virtual void draw() {
                printf("Point: (%d, %d)\n", _x, _y);
        }
};
```

A função virtual draw da classe Point2D imprime no ecrã as coordenadas do eixo dos xx (_x) e do eixo dos yy (_y) de um ponto a duas dimensões. Como a função membro draw da classe Point2D é definida como virtual e a classe Point3D herda da classe Point2D, a função membro draw da classe Point2D é reimplementada pela classe Point3D:

```
class Point3D : public Point2D {
        int _z;
public:
        ...
        void draw(){
                printf("Point: (%d, %d, %d)\n", _x, _y, _z);
        }
};
```

A classe Point3D ao reimplementar o método draw da classe Point2D, imprime no ecrã as coordenadas do eixo dos xx (_x), do eixo dos yy (_y) e do eixo dos zz (_z) de um ponto a três dimensões.

O código seguinte mostra que independente de se executar a função virtual draw de um ponto a três dimensões, através de uma instância do ponto do tipo Point3D ou do tipo Point2D, é sempre executada a função membro draw da classe Point3D:

```
Point3D *point3D = new Point3D(1, 2, 3); /* cria um objeto do tipo Point3D */
Point2D *point2D = point3D; /* faz a conversão de apontador Point3D para Point2D */
point3D->draw(); /* chama a função membro draw da classe Point3D */
point2D->draw(); /* chama a função membro draw da classe Point3D */
```

Para explicar o facto de ser possível chamar o método draw da classe Point3D através de uma instância do ponto do tipo Point2D, é apresentado na figura 3.4 a forma como os dados da classe Point3D estão organizados em memória:

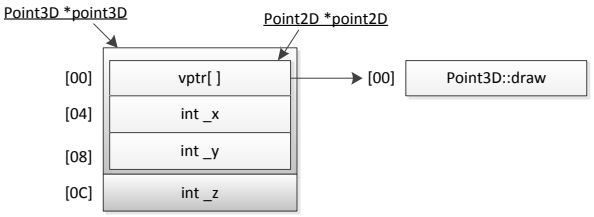


Figura 3.4: Disposição dos dados da classe Point3D em memória.

Sempre que uma classe define uma função virtual, os primeiros 4 *bytes* dos dados da classe contêm o endereço de memória da tabela virtual (vptr). A tabela virtual é um *array* de apontadores para funções, onde cada um dos elementos do *array* aponta para as funções virtuais definidas na classe. Assim, no exemplo apresentado o *array* da tabela virtual é constituído apenas por um elemento que aponta para a função membro init da classe Point3D. Logo, independentemente de se utilizar um apontador do tipo Point3D ou do tipo Point2D, é sempre executada a função que está no índice zero do *array* da tabela virtual.

Implementação em C

Para implementar a mesma funcionalidade em C, é necessário definir a estrutura de dados Point2D:

```
struct Point2D {
        void **vptr;
        int _x;
        int _y;
};
```

A estrutura de dados Point2D tem que definir as variáveis que permitam obter a mesma organização dos dados da classe Point2D presente na figura 3.4. Assim, os primeiros 4 *bytes* são reservados para a tabela virtual (vptr), os 4 *bytes* seguinte para a coordenada do ponto no eixo dos xx (_x) e os 4 *bytes* restantes para a coordenada do ponto no eixo dos yy (_y).

Ao definir a estrutura de dados Point2D, os primeiros 4 *bytes* da estrutura são reservados para a variável vptr, que é um duplo apontador para o tipo de dados void. O objetivo da variável vptr é apontar para um *array* de apontadores para funções,

onde cada um dos elementos do *array* aponta para as funções virtuais definidas na classe Point2D. Para inicializar o *array* de apontadores de funções apontado pela variável vptr da estrutura de dados Point2D, é necessário criar e definir cada uma das funções virtuais do *array*. O código seguinte, cria e inicializa o *array* Point2D_VPTR:

```
/* define o protótipo da função virtual Point2D_Draw */
void Point2D_Draw(struct Point2D *this);

/* define e inicializa a tabela virtual (array de funções virtuais) */
void* Point2D_VPTR[] = {
    (void*)&Point2D_Draw
};
```

O *array* de funções virtuais Point2D_VPTR é constituído apenas por um elemento que aponta para a função virtual Point2D_Draw. De seguida é apresentada a função construtor da estrutura Point2D:

```
void Point2D_Constructor(struct Point2D *this, int x, int y) {
    this->vptr = Point2D_VPTR;
    this->_x = x;
    this->_y = y;
}
```

A função construtor da estrutura Point2D recebe como parâmetros de entrada: o apontador this que aponta para os dados da estrutura Point2D, a variável x que contém o valor da coordenada do eixo dos xx e a variável y que contém o valor da coordenada do eixo dos yy. O corpo da função define a tabela virtual vptr e as coordenadas do ponto com os valores x e y.

Por fim, é necessário implementar a função virtual da estrutura Point2D:

```
void Point2D_Draw(struct Point2D *this) {
    printf("Point: (%d, %d)\n", _x, _y);
}
```

A função virtual Point2D_Draw recebe como parâmetro de entrada o apontador this que aponta para o ponto, e imprime no ecrã o valor das variáveis membros _x e _y do ponto.

O código seguinte, cria uma estrutura do tipo Point2D e executa a função virtual Point2D_Draw da estrutura:

```
struct Point2D point2D;
Point2D_Constructor(&point2D, 1, 2);
point2D.vptr[0](&point2D);
```

Para chamar a função virtual Point2D_Draw da estrutura Point2D, é executado o apontador para função da tabela virtual vptr com índice zero.

De seguida é apresentado o código que define a estrutura de dados Point3D:

```
struct Point3D {
        struct Point2D pt;
        int _z;
};
```

Como em C++ a classe Point3D herda da classe Point2D, a primeira variável da estrutura de dados Point3D tem que ser do tipo Point2D. Para definir a componente do eixo dos zz, é definida a variável _z na estrutura de dados Point3D.

Depois, é necessário definir o protótipo da função virtual Point2D_Draw e o *array* de funções virtuais:

```
/* define o protótipo da função virtual Point3D_Draw */
void Point3D_Draw(struct Point3D *this);

/* define e inicializa a tabela virtual (array de funções virtuais) */
void* Point3D_VPTR[] = {
        (void*)&Point3D_Draw
};
```

O protótipo da função virtual Point3D_Draw define que a função recebe um parâmetro de entrada que é um apontador para uma estrutura do tipo Point3D e devolve void. O *array* de funções virtuais Point3D_VPTR é constituído apenas por um elemento que aponta para a função virtual Point3D_Draw.

De seguida é apresentada a função construtor da estrutura Point3D:

```
void Point3D_Constructor(struct Point3D *this, int x, int y, int z) {
        Point2D_Constructor((struct Point2D*)this, x, y);
        this->pt.vptr = Point3D_VPTR;
        this->_z = z;
}
```

A função construtor da estrutura Point3D recebe como parâmetros de entrada: o apontador this que aponta para os dados da estrutura Point3D, a variável x que contém o valor da coordenada do eixo dos xx, a variável y que contém o valor da coordenada do eixo dos yy e a variável z que contém o valor da coordenada do eixo dos zz. A função chama o construtor da estrutura Point2D para inicializar a tabela virtual e as coordenadas x e y do ponto a duas dimensões. Depois, é inicializada a tabela virtual vptr do ponto a duas dimensões com o vetor Point3D_VPTR e a coordenada do eixo dos zz.

Por fim, é necessário implementar a função virtual da estrutura Point3D:

```
void Point3D_Draw(struct Point3D *this) {
        printf("Point: (%d, %d)\n", this->_x, this->_y);
}
```

A função virtual Point3D_Draw recebe como parâmetro de entrada o apontador this que aponta para o ponto, e imprime no ecrã o valor das variáveis membros _x, _y e _z do ponto.

O código seguinte, cria uma estrutura do tipo Point3D e executa a função virtual Point3D_Draw da estrutura:

```
struct Point3D point3D;
struct Point2D *point2D;
Point3D_Constructor(&point3D, 1, 2, 3);
point2D = &point3D.pt;
point3D.pt.vptr[0](&point2D);
point2D->vptr[0](&point3D);
```

Ao analisar o código das duas implementações, verifica-se que a implementação em C++ utilizando herança e funções virtuais é muito mais fácil de implementar, ocupa menos linhas de código e é muito mais percetível que a mesma implementação em C utilizando estruturas de dados.

Código *Assembly*

Para verificar se as funções virtuais produzem algum custo ao nível do desempenho e do tamanho do ficheiro executável, é apresentado o código *assembly* gerado pelo compilador C++ para o construtor da classe Point3D:

```
??0Point3D@@QAE@HHH@Z proc near
    ...
    mov eax, [ebp+y] ; o registo eax toma o valor do parâmetro de entrada y
    push eax ; coloca na pilha o valor de y
    mov ecx, [ebp+x] ; o registo ecx toma o valor do parâmetro de entrada x
    push ecx ; coloca na pilha o valor de x
    mov ecx, [ebp+this] ; o registo ecx toma o valor do apontador this da classe
    call ??0Point2D@@QAE@HH@Z ; chama o construtor da classe Point2D
    mov edx, [ebp+this] ; o registo edx toma o valor do apontador this da classe
    mov dword ptr [edx], offset ??_7Point3D@@6B@ ; inicializa a tabela virtual
    mov eax, [ebp+this] ; o registo eax toma o valor do apontador this da classe
    mov ecx, [ebp+z] ; o registo ecx toma o valor do parâmetro de entrada z
    mov [eax+0Ch], ecx ; a variável membro _z da classe Point3D toma o valor de z
    ...
??0Point3D@@QAE@HHH@Z endp
```

O código *assembly* gerado pelo compilador C++ para o construtor da classe Point3D é muito semelhante ao código *assembly* gerado pelo compilador C para a função que inicializa um ponto do tipo Point3D.

De seguida, é apresentado o código *assembly* que executa a função virtual draw da classe Point3D:

```
...
mov edx, [ebp+point3D] ; o registo edx toma o valor do apontador this da classe
mov eax, [edx] ; o registo eax toma o valor do array de funções apontadores da tabela virtual
mov ecx, [ebp+point3D] ; o registo ecx toma o valor do apontador this da classe
mov edx, [eax] ; o registo edx toma o valor do 1º item do array de funções apontadores
call edx ; chama a função virtual draw da classe Point3D
...
```

Ao analisar o código *assembly* anterior, verifica-se que sempre que é executada uma função virtual é necessário: (i) ler o endereço da tabela virtual, (ii) ler o endereço da função apontador com índice `n` da tabela virtual (função virtual a executar) e (iii) chamar a função usando o modo de endereçamento direto por registo.

Portanto, os principais problemas associados às funções virtuais são: (i) as funções virtuais fazem com que a memória necessária para alocar um objeto seja maior (4 *bytes* para a tabela virtual), (ii) sempre que uma função virtual é executada é necessário ler o endereço de memória da função a executar da tabela virtual, em vez de esta ser executada diretamente e (iii) as funções virtuais de qualquer classe têm que ser resolvidas pelo *linker* mesmo que estas não sejam executadas na aplicação.

Assim, conclui-se que a utilização de funções virtuais produz um impacto negativo no desempenho e no tamanho do ficheiro executável da aplicação.

3.3.10 *Templates*

As *templates* permitem criar funções e classes que operam com tipos de dados genéricos. Isto é, quando existem múltiplas cópias do mesmo código que implementam a mesma lógica de implementação para diferentes tipos de dados, devem-se utilizar *templates*. As *templates* são muito semelhantes às macros, mas permitem implementar código mais seguro estática e semanticamente. Assim, é possível escrever menos código e o processo de manutenção do código é muito mais fácil de gerir.

O código seguinte mostra um exemplo de uma classe *template*:

```
template <typename T>
class Point2D {
      T _x, _y;
public:
      Point2D(T x, T y)
         : _x(x), _y(y) {}
};
```

A classe *template* Point2D cria um ponto a duas dimensões genérico, onde é possível definir qual o tipo de dados das coordenadas do ponto.

O código seguinte, apresenta um exemplo da instanciação de dois pontos com tipos de dados diferentes:

```
Point2D<int> point2D_int(1, 2);
Point2D<float> point2D_float(1.50, 2.50);
```

O código apresentado define dois objetos: o objeto point2D_int e o objeto point2D_float. O objeto point2D_int cria um ponto a duas dimensões onde as coordenadas do ponto são do tipo inteiro. O objeto point2D_float cria um ponto a duas dimensões onde as coordenadas do ponto são do tipo vírgula flutuante. No ato da compilação, o compilador C++ gera de forma automática uma classe baseada na *template* para o tipo de dados int e para o tipo de dados float, onde o parâmetro de entrada T da *template* Point2D é substituído pelo tipo de dados correspondente.

Implementação em C

Para implementar a mesma funcionalidade em C, é necessário numa primeira fase definir a estrutura de dados Point2D_Int, onde as coordenadas do ponto são do tipo inteiro:

```
struct Point2D_Int {
    int _x;
    int _y;
};
```

Depois é necessário definir a função que inicializa os dados de um ponto do tipo Point2D_Int:

```
void Point2D_Constructor_Int(struct Point2D_Int *this, int x, int y) {
    this->_x = x;
    this->_y = y;
}
```

Numa segunda fase, é necessário definir a estrutura de dados Point2D_Float, onde as coordenadas do ponto são do tipo vírgula flutuante:

```
struct Point2D_Float {
    float _x;
    float _y;
};
```

Depois é necessário definir a função que inicializa os dados de um ponto do tipo Point2D_Float:

```
void Point2D_Constructor_Float(struct Point2D_Float *this, float x, float y) {
    this->_x = x;
    this->_y = y;
}
```

O código seguinte apresenta um exemplo da definição de dois pontos, onde as coordenadas dos pontos são de tipos diferentes:

```
struct Point2D_Int point2D_int;
struct Point2D_Float point2D_float;

Point2D_Constructor_Int(&point2D_int, 1, 2);
Point2D_Constructor_Float(&point2D_float, 1.50, 2.50);
```

Ao analisar as duas implementações, verifica-se que a implementação em C é muito mais complexa que a implementação em C++ com *templates*. Ao utilizar *templates* em C++, é o compilador que cria de forma automática uma cópia da definição da classe Point2D para cada um dos tipos de dados (int e float). Em C, é o programador que necessita de criar a estrutura de dados Point2D para cada um dos tipos de dados (int e float).

Código *Assembly*

Para verificar se as *templates* em C++ produzem algum custo ao nível do desempenho e do tamanho do ficheiro executável, é apresentado o código *assembly* gerado pelo compilador C++ para o construtor das duas *templates* apresentadas no exemplo anterior.

O código *assembly* seguinte diz respeito ao construtor da classe Point2D quando esta é instanciada com o tipo de dados int:

```
??0?$Point2D@H@@QAE@HH@Z proc near
    ...
    mov eax, [ebp+this] ; o registo eax toma o valor do apontador da classe
    mov ecx, [ebp+x] ; o registo ecx toma o valor do parâmetro de entrada x
    mov [eax], ecx ; a variável membro _x da classe Point2D toma o valor de x
    mov edx, [ebp+this] ; o registo edx toma o valor do apontador da classe
    mov eax, [ebp+y] ; o registo eax toma o valor do parâmetro de entrada y
    mov [edx+4], eax ; a variável membro _y da classe Point2D toma o valor de y
    ...
??0?$Point2D@H@@QAE@HH@Z endp
```

O código *assembly* seguinte diz respeito ao construtor da classe Point2D quando esta é instanciada com o tipo de dados float:

```
??0?$Point2D@M@@QAE@MM@Z proc near
    ...
```

```
        mov eax, [ebp+this] ; o registo eax toma o valor do apontador da classe
        fld [ebp+x] ; coloca na pilha o valor do parâmetro de entrada x
        fstp dword ptr [eax] ; a variável membro _x da classe Point2D toma o valor de x lido da pilha
        mov ecx, [ebp+this] ; o registo ecx toma o valor do apontador da classe
        fld [ebp+y] ; coloca na pilha o valor do parâmetro de entrada y
        fstp dword ptr [ecx+4] ; a variável membro _y da classe Point2D toma o valor de y lido da pilha
        ...
??0?$Point2D@M@@QAE@MM@Z endp
```

Ao analisar o código *assembly* gerado pelo compilador C++ para os construtores das duas classes *template*, verifica-se que o código gerado é similar ao código gerado para as mesmas funções da versão em C.

Assim, conclui-se que a utilização de *templates* não apresenta custos no desempenho ou no aumento do tamanho do ficheiro executável de uma aplicação.

3.3.11 Exceções

As exceções fornecem um mecanismo de tratamento de erros que separa o código normal do código responsável pelo tratamento de erros, permitindo assim que no caso de ocorrer um erro o controlo do programa seja transferido para funções especias às quais se dá o nome de *handlers*. Para que os erros de um determinado código seja tradado/gerido através de exceções, é necessário que o código esteja dentro de um bloco *try-and-catch*. Assim, se algum erro ocorrer durante a execução do código, é executado o *handler* que processa a exceção que desencadeou o erro.

O código seguinte apresenta o operador /= da classe Point2D:

```
const Point2D& Point2D::operator/=(const Point2D& pt) {
    try {
        _x /= pt._x;
        _y /= pt._y;
        return *this;
    } catch (std::exception ex) {
        // processa o erro
    }
}
```

O operador /= da classe Point2D recebe como parâmetro de entrada a referência pt do tipo Point2D e divide as coordenadas do ponto em questão (this) pelas coordenadas do ponto pt. O código da divisão é colocada dentro de um bloco *try-and-catch* porque a divisão por zero provoca uma exceção.

Implementação em C

Para implementar as exceções em linguagem de programação C, é necessário em primeiro lugar entender a forma como o compilador C++ implementa as exceções em código *assembly*. Como o *standard* C++ não especifica a forma como as exceções devem ser implementadas, cada compilador implementa as exceções à sua maneira. Tal como foi referido anteriormente, para compilar o código C++ dos exemplos apresentados, foi utilizado o compilador Microsoft Visual C++. Logo, é apenas explicada a forma como o compilador Microsoft Visual C++ implementa as exceções em código *assembly*.

Existem dois tipos de exceções que podem ser utilizadas no compilador C++ da Microsoft: (i) exceções SEH (*Structured Exception Handling*) e (ii) exceções C++ (muitas vezes conhecidas por EH). As exceções SEH são fornecidas pelo próprio sistema operativo Windows. As exceções C++ são implementadas no topo do SEH, permitindo extender o SEH de forma a que possam ser processados quaisquer tipo de exceções e ser feito o *unwinding* automático da pilha quando uma exceção é processada. Neste capítulo são explicadas apenas as exceções SEH, visto serem estas a base de implementação das exceções C++.

Tal como foi referido anteriormente, as exceções SEH são fornecidas pelo sistema operativo Windows. Para registar cada uma das exceções, o Windows define a estrutura de dados EXCEPTION_REGISTRATION:

```
struct EXCEPTION_REGISTRATION {
    EXCEPTION_REGISTRATION *prev;
    DWORD handler;
};
```

A estrutura de dados EXCEPTION_REGISTRATION é composta por duas variáveis: a variável prev do tipo apontador para a estrutura de dados EXCEPTION_REGISTRATION e a variável handler do tipo DWORD. A variável prev aponta para o nó anterior da lista ligada de exceções. A variável handler define o endereço de memória da função que é responsável pelo tratamento da exceção.

Para criar uma nova exceção, é necessário criar uma nova estrutura de dados para registar a exceção (estrutura do tipo EXCEPTION_REGISTRATION), preencher a estrutura com os dados corretos e guardar o endereço de memória da estrutura no *offset* zero do segmento FS do x86, isto é FS:[0].

O código seguinte permite obter o primeiro elemento da lista ligada de exceções:

```
DWORD prev; ; variável que contém o topo da lista ligada de exceções
_asm {
    mov eax, fs:[0] ; o registo eax toma o valor do endereço de memória do topo da lista
    mov prev, eax ; guarda na variável prev o endereço de memória do topo da lista
}
```

Depois de a variável prev apontar para o topo da lista ligada de exceções, é necessário definir a função Div_Handler que processa a exceção da divisão por zero:

```
EXCEPTION_DISPOSITION Div_Handler(_EXCEPTION_RECORD *ExcRecord,
        void * EstablisherFrame, _CONTEXT *ContextRecord, void * DispatcherContext)
{
    // processa o erro
}
```

Todos os *handlers* de exceções têm quatro parâmetros de entrada, tal como os definidos no *handler* Div_Handler. O propósito de cada um dos parâmetros de entrada dos *handlers* não é explicado neste documento porque não é relevante.

Depois de definido o *handler* Div_Handler, é criada uma nova estrutura de dados do tipo EXCEPTION_REGISTRATION para registar a nova exceção:

```
EXCEPTION_REGISTRATION reg;
EXCEPTION_REGISTRATION *preg = &reg;
...
reg.prev = (EXCEPTION_REGISTRATION *) prev;
reg.handler = (DWORD) Div_Handler;
```

A variável prev da estrutura de dados que regista a nova exceção é definida com o valor do topo da lista ligada de exceções e a variável handler com o endereço de memória da função Div_Handler.

Por fim, é necessário registar a nova lista ligada de exceções nos primeiros quatro *bytes* do TIB indexadas no segmento FS do x86, isto é FS:[0]:

```
_asm {
    mov eax, preg; o registo eax toma o valor do endereço de memória da nove estrutura
    mov fs:[0], eax ; os primeiros 4 bytes do registo FS tomam o valor do endereço de
                    ; memória do topo da lista
}
```

Do código apresentado, verifica-se que as exceções utilizam de forma intensiva a pilha, isto é, sempre que é encontrado um novo bloco *try-and-catch* no código da aplicação, é necessário criar uma nova estrutura na pilha para definir a nova exceção, definir a nova exceção com os valores corretos e por fim, os primeiros 4 *bytes* endereçadas a partir do registo FS têm que apontar para a nova estrutura, que representa o novo topo da lista de exceções. No final do bloco *try-and-catch*

é necessário fazer a operação inversa, para definir o novo topo da lista como o anterior e fazer o *unwind* da pilha.

Código *Assembly*

Para verificar se as exceções produzem algum custo ao nível do desempenho e do tamanho do ficheiro executável, é apresentado o código *assembly* gerado pelo compilador C++ do operador /= da classe Point2D:

```
??_0Point2D@@QAEABV0@ABV0@@Z proc near
    ...
    push 0FFFFFFFFh ; coloca o valor NULL na pilha
    push offset ___ehhandler$??_0Point2D@@QAEABV0@ABV0@@Z ; coloca na pilha
                                ; o endereço do novo handler responsável pelo tratamento da exceção
    mov eax, large fs:0 ; o registo eax toma o valor do primeiro endereço do registo FS
    push eax ; guarda o valor na pilha
    ...
    mov large fs:0, ecx ; define como novo topo da lista ligada de exceções o nó anterior da lista
    ...
??_0Point2D@@QAEABV0@ABV0@@Z endp
```

Ao analisar o código *assembly* gerado pelo compilador C++ do operador /= da classe Point2D, conclui-se que as exceções utilizam de forma intensiva a pilha no tratamento de exceções. Assim, conclui-se que as exceções são processadas pelo *back-end* fazendo com que diminua o desempenho e aumente o tamanho do ficheiro executável da aplicação.

3.3.12 RTTI

O RTTI (*Run-Time Type Information*) é um mecanismo que permite obter a informação do tipo de dados de um objeto em tempo de execução. O operador dynamic_cast<> e o operador typeid utilizam o mecanismo RTTI, que pode apenas ser utilizado em classes polimórficas, isto é, em classes que definem pelo menos um método virtual.

Para demonstrar como utilizar o mecanismo RTTI, é dado um exemplo da conversão dinâmica de dados de um apontador do tipo Point2D para um apontador do tipo Rect2D utilizando o operador dynamic_cast<>. No exemplo é utilizada a classe Point2D definida anteriormente na secção das funções virtuais e a classe Rect2D que implementa um rectângulo:

```
class Rect2D {
    int _x, _y, _w, _h;
```

```
public:
        Rect2D(int x, int y, int w, int h)
                : _x(x), _y(y), _w(w), _h(h) {}

        virtual void draw() {
                printf("(%d, %d, %d, %d)\n", _x, _y, _w, _h);
        }
};
```

Como ambas as classes são polimórficas, podem ser utilizadas em funcionalidades que utilizam o mecanismo RTTI, mais propriamente no operador dynamic_cast<>.

O código seguinte faz a conversão dinâmica de dados de um apontador do tipo Point2D para um apontador do tipo Rect2D:

```
Point2D *point2D = new Point2D(1, 2);
Rect2D *rect2D = dynamic_cast<Rect2D*>(point2D);
if (rect2D == NULL) {
        printf("A conversão entre os dois tipos de dados não é possível!\n");
}
```

Para fazer a conversão dinâmica entre um apontador do tipo de dados Point2D e um apontador do tipo de dados Rect2D, é utilizado o operador dynamic_cast<>. Se a conversão entre os dois tipos de dados não for possível, o apontador rect2D aponta para NULL.

Implementação em C

Como o mecanismo RTTI não é suportado em linguagem C, é necessário implementar de raiz as estruturas de dados e funções que dão suporte ao RTTI, mais especificamente ao operador de conversão dinâmica de dados dynamic_cast<>.

Assim, é necessário definir a estrutura de dados Point2D e implementar a função Point2D_Draw. Como a estrutura de dados Point2D e a função Point2D_Draw já foram definidas na secção das funções virtuais, não é necessário as definir novamente.

O código seguinte, define a estrutura de dados Rect2D:

```
struct Rect2D {
        void **vptr;
        int _x;
        int _y;
        int _w;
        int _h;
};
```

A estrutura de dados Rect2D, define primeiro a variável vptr que representa a tabela virtual da classe e de seguida as variáveis membros da classe Rect2D. A ordem pela qual os dados da estrutura Rect2D estão dispostos tem que seguir a mesma ordem em que os dados da classe Rect2D estão dispostos em memória.

Para inicializar o *array* de apontadores de funções apontado pela variável vptr da estrutura de dados Rect2D, é necessário criar e definir cada uma das funções virtuais do *array*. O código seguinte, cria e inicializa o *array* Rect2D_VPTR:

```
/* define o protótipo da função virtual Rect2D_Draw */
void Rect2D_Draw(struct Rect2D *this);

/* define e inicializa a tabela virtual (array de funções virtuais) */
void* Rect2D_VPTR[] = {
        (void*)&Rect2D_Draw
};
```

O *array* de funções virtuais Rect2D_VPTR é constituído apenas por um elemento que aponta para a função virtual Rect2D_Draw.

De seguida é apresentada a função construtor da estrutura Rect2D:

```
void Rect2D_Constructor(struct Rect2D *this, int x, int y, int w, int h) {
        this->vptr = Rect2D_VPTR;
        this->_x = x;
        this->_y = y;
        this->_w = w;
        this->_h = h;
}
```

A função construtor da estrutura Rect2D recebe como parâmetros de entrada: o apontador this que aponta para os dados da estrutura Rect2D, a variável x que contém o valor inicial do rectângulo no eixo dos xx, a variável y que contém o valor inicial do rectângulo no eixo dos yy, a variável w que contém o valor do comprimento do rectângulo e a variável h que contém o valor da altura do rectângulo. A função construtor Rect2D_Constructor, inicializa a tabela virtual e as variáveis da estrutura referentes ao rectângulo.

Depois de definidas as estruturas Point2D e Rect2D, é necessário criar as estruturas que servem de suporte ao operador dynamic_cast<> do RTTI. Assim, numa primeira fase é definida a estrutura de dados TypeDescriptor:

```
struct TypeDescriptor {
        DWORD *pVFTable;
        DWORD spare;
        char *name;
};
```

A estrutura de dados TypeDescriptor contém a informação sobre uma determinada classe em tempo de execução, tal como, um apontador para a tabela com a informação do tipo de dados (type_info) e o nome da classe.

O código seguinte define o TypeDescriptor da classe Point2D:

```
struct TypeDescriptor Point2D_TypeDescriptor = {
        &Point2D_VFTable_TypeInfo,
        0,
        "Point2D"
};
```

Para definir a informação do TypeDescriptor da classe Rect2D são necessárias as seguintes linhas de código:

```
struct TypeDescriptor Rect2D_TypeDescriptor = {
        &Point2D_VFTable_TypeInfo,
        0,
        "Rect2D"
};
```

Por fim, é apresentado o código que permite fazer a conversão dinâmica de dados de um apontador para uma estrutura do tipo Point2D para um apontador para uma estrutura do tipo Rect2D:

```
...
struct Point2D point2D;
struct Rect2D *rect2D;
Point2D_Construtor(&point2D, 1, 2);
rect2D = RTDynamicCast(&point2D, 0,
            Point2D_TypeDescriptor, Rect2D_TypeDescriptor, 0);
if (rect2D == NULL) {
        printf("A conversão entre os dois tipos de dados não é possível!\n");
}
```

Na conversão é utilizada a API do Windows responsável pela conversão dinâmica de dados em tempo de execução RTDynamicCast. A função RTDynamicCast recebe cinco parâmetros de entrada: O apontador para o objeto a converter (&point2D), o *offset* da tabela virtual do objeto, a informação estática do TypeDescriptor da classe Point2D, a informação estática do TypeDescriptor da classe Rect2D e a informação que o objeto a converter é um apontador.

Ao analisar o código das duas implementações, verifica-se que o código C++ é muito mais simples e percetível que o código C.

Código *Assembly*

Para verificar se o operador dynamic_cast<> do mecanismo RTTI produz algum custo ao nível do desempenho e do tamanho do ficheiro executável, é apresentado o código *assembly* da conversão dinâmica de dados de um apontador do tipo Point2D para um apontador do tipo Rect2D gerado pelo compilador C++:

```
...
push 0 ; indica que o tipo de dados a converter é um apontador
push offset ??_R0?AVRect2D@@@8 ; Rect2D 'RTTI Type Descriptor'
push offset ??_R0?AVPoint2D@@@8 ; Point2D 'RTTI Type Descriptor'
push 0 ; offset da tabela virtual do objeto Point2D
mov edx, [ebp+point2D] ; o registo edx contém o endereço de memória do objeto point2D
push edx ; passa o objeto point2D como parâmetro de entrada
call ___RTDynamicCast
...
```

Os dados da estrutura TypeDescriptor da classe Rect2D são apresentados a seguir:

```
??_R0?AVRect2D@@@8 dd offset ??_7type_info@@6B@ ; TypeDescriptor.pVFTable
                   dd 0 ; TypeDescriptor.sparce
                   db '.?AVRect2D@@',0 ; TypeDescriptor.name
```

E os dados da estrutura TypeDescriptor da classe Point2D são apresentados a seguir:

```
??_R0?AVPoint2D@@@8 dd offset ??_7type_info@@6B@ ; TypeDescriptor.pVFTable
                    dd 0 ; TypeDescriptor.sparce
                    db '.?AVPoint2D@@',0 ; TypeDescriptor.name
```

Ao analisar o código assembly gerado pelo compilador, conclui-se que para que seja possível obter informação sobre o tipo de dados de um objeto em tempo de execução, é necessário introduzir estruturas em *assembly* com a informação de cada um dos objetos e fazer a conversão dos dados utilizando a API do Windows RTDynamicCast. Assim, conclui-se que o mecanismo RTTI é uma funcionalidade do *back-end* do compilador C++, fazendo com que diminua o desempenho e aumente o tamanho do ficheiro executável da aplicação.

3.4 Conclusão

Neste capítulo foi apresentada a linguagem de programação C++ como a linguagem de eleição para o desenvolvimento de aplicações para sistemas embebidos. Contrariamente ao que a maioria dos programadores de aplicações para sistemas embebidos pensam, a maioria das funcionalidades da linguagem de programação

C++ não apresentam qualquer tipo de custo ao nível do desempenho e do tamanho do ficheiro executável das aplicações comparativamente com a linguagem de programação C.

Ao desenvolver aplicações para sistemas embebidos em C++, o programador beneficia a três níveis: (i) o *type checking* do compilador C++ é mais poderoso que o do compilador C, fazendo com que hajam menos erros de programação, (ii) o código *assembly* gerado pelo compilador C++ é mais otimizado que o código *assembly* gerado pelo compilador C e (iii) como a linguagem de programação C++ é uma linguagem orientada ao objeto, disponibiliza funcionalidades que permitem desenvolver e gerir o código das aplicações de uma forma mais fácil e segura que o código C, principalmente quando a complexidade do sistema a implementar aumenta.

Assim, das funcionalidades abordadas, as que são processadas pelo *front-end* do compilador C++ não produzem qualquer impacto no desempenho e no tamanho do ficheiro executável das aplicações. Destas, destacam-se: as classes, os *namespaces*, as funções *inline*, o *overloading* de operadores, o construtor/destrutor, as referências, a herança única e as *templates*. As funcionalidades que são processadas pelo *back-end* do compilador C++, produzem impacto no desempenho e no tamanho do ficheiro executável das aplicações. As funcionalidades que são processadas pelo *back-end* do compilador C++ são: a múltipla herança, as funções virtuais, as exceções e o RTTI.

Capítulo 4

DirectShow em Sistemas Embebidos

4.1 Introdução

No capítulo anterior, foram enumeradas as funcionalidades da linguagem de programação C++ que penalizam o tamanho do código binário e o tempo de execução das aplicações desenvolvidas para serem executadas em sistemas embebidos. Das funcionalidades enumeradas, destacam-se: o polimorfismo dinâmico, a múltipla herança, o tratamento de exceções, a informação do tipo de dados em tempo de execução (RTTI) e a conversão dinâmica de dados (*dynamic cast*).

Neste capítulo, são apresentados os conceitos básicos do *DirectShow*, a forma como cada um dos componentes do *DirectShow* é implementado e o impacto que cada um dos componentes provoca nas aplicações que são compiladas para serem executadas em sistemas embebidos. Assim, numa primeira fase, é feita uma breve descrição da arquitetura e dos principais componentes do *DirectShow* (*filter graphs*, filtros e pinos). De seguida, é abordada a forma como os vários componentes da *framework* são implementados, dando especial atenção às funcionalidades da linguagem de programação C++ que são utilizadas na implementação dos vários componentes. Por fim, são apresentadas as funcionalidades da linguagem de programação C++ que são utilizadas na implementação do *DirectShow*, que afectam de forma negativa o tamanho do código binário e o tempo de execução das aplicações *DirectShow*, em sistemas embebidos.

4.2 Conceitos Básicos do *DirectShow*

O *DirectShow* é uma *framework* multimédia desenvolvida pela Microsoft para facilitar o desenvolvimento de aplicações multimédia. A *framework* disponibiliza uma API baseada no *Component Object Model* (COM), que permite desenvolver aplicações *DirectShow* utilizando diferentes linguagens de programação.

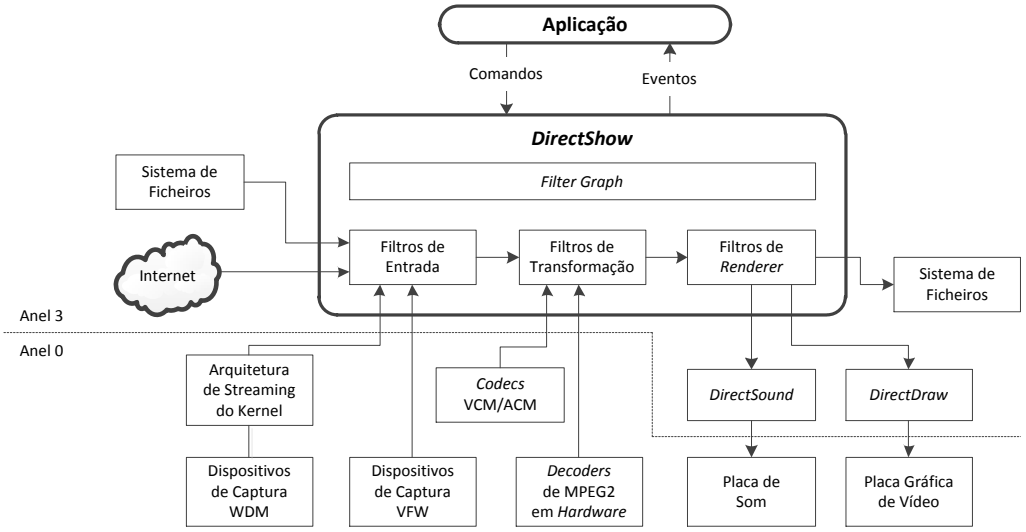


Figura 4.1: Relação entre uma aplicação *DirectShow* e os vários componentes do *DirectShow*. Imagem adaptada de Pesce (2002).

O diagrama da figura 4.1, apresenta a relação que existe entre uma aplicação *DirectShow* e os vários componentes do *DirectShow*. Facilmente se percebe que o *DirectShow* é constituído por um conjunto de componentes, aos quais se dá o nome de filtros. Os filtros formam o núcleo da *framework* e são classificados em função do seu tipo: filtros de entrada, filtros de transformação e filtros de *rendering*.

O *filter graph* é um filtro especial que agrega um conjunto de filtros para definir o fluxo de execução de dados da aplicação *DirectShow*. Uma vez criados os *filter graphs* da aplicação *DirectShow*, a única tarefa que a aplicação necessita de realizar é enviar comandos para os *filter graphs* e processar os eventos enviados pelos *filter graphs*. Desta forma, torna-se evidente que a utilização do *DirectShow* simplifica a criação de aplicações multimédia, isolando a aplicação da complexidade inerente ao desenvolvimento de cada um dos componentes da *framework*.

4.2.1 Arquitetura Modular

A *framework* multimédia *DirectShow*, assenta numa arquitetura modular baseada em filtros sob a forma de *plugins*. Este tipo de arquitetura, apresenta como principal vantagem, o facto de permitir uma maior flexibilidade, ao nível da gestão e da extensibilidade do código fonte do sistema, em tempo de compilação e uma maior flexibilidade, ao nível das várias configurações possíveis do sistema, em tempo de execução.

Tal como foi referido anteriormente, a *framework* é constituída por um conjunto de filtros, onde cada um implementa uma determinada funcionalidade. Sempre que seja necessário extender a *framework*, é criado um novo filtro que implementa a funcionalidade pretendida. Os filtros são implementados sob a forma de *plugins* em bibliotecas dinâmicas (DLLs) e registados na *framework* de forma automática.

4.2.2 Principais Componentes do *DirectShow*

O *DirectShow* é composto por três componentes principais: os *filter graphs*, os filtros e os pinos. Os três componentes mantêm uma relação de composição, onde cada um dos *filter graphs* é composto por um conjunto de filtros e por sua vez cada um dos filtros pode conter um conjunto de pinos.

Filter Graph

O *filter graph* é a entidade que representa o fluxo de execução de dados de uma aplicação *DirectShow*. Mais precisamente, o *filter graph* representa uma unidade funcional que é constituída por um conjunto de componentes (filtros), que estão ligados entre si. As ligações entre os vários filtros do *filter graph*, definem o fluxo de execução de dados da aplicação *DirectShow*, o qual segue o modelo de execução em *pipeline*. Desta forma, como os filtros estão ligados uns aos outros, os dados são transferidos de uns para os outros de forma sequencial.

A figura 4.2 apresenta um exemplo de uma configuração de um *filter graph* constituído por cinco filtros. As ligações entre os vários filtros definem o fluxo de execução de dados do *filter graph*. Para se obter uma nova configuração do *filter graph* é necessário adicionar ou remover filtros ao *filter graph* e estabelecer novas ligações entre eles. Esta operação pode ser realizada em tempo de execução, visto

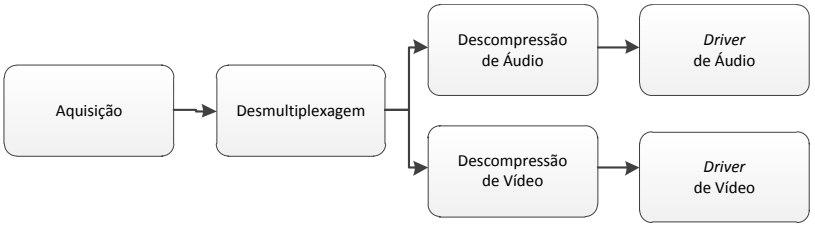


Figura 4.2: Exemplo de uma configuração do *filter graph*.

que o fluxo de execução de dados do *filter graph* foi desenhado para ser alterado dinamicamente.

Embora o *filter graph* implemente funções que permitem adicionar e remover filtros, e estabelecer as ligações entre os vários filtros, também implementa funções de controlo. As funções de controlo mais importantes são as que permitem iniciar, fazer pausa e parar o fluxo de execução de dados do *filter graph*.

Stream de Dados Sempre que uma aplicação *DirectShow* necessite de implementar mais do que um fluxo de execução de dados, pode-o fazer de duas formas: utilizando vários *filter graphs* ou utilizando um só *filter graph*. A forma mais comum de implementar é utilizando vários *filter graphs*, em que cada um contém apenas uma *stream* de dados. Caso seja utilizado um só *filter graph*, são definidas várias *streams* de dados dentro do *filter graph*, cada uma delas representando um dos fluxos de execução de dados. A título de exemplo, o *filter graph* da figura 4.2 é constituído apenas por uma *stream* de dados.

Filtros

Os filtros são as unidades elementares da *framework* multimédia *DirectShow*. O fluxo de execução de qualquer aplicação *DirectShow* (*filter graph*) é constituído por um conjunto de filtros. Cada filtro representa uma entidade indivisível e implementa uma funcionalidade que é independente das funcionalidades implementadas pelos restantes filtros do *filter graph* e da própria *framework*.

A aplicação *DirectShow* da figura 4.2 é constituída por cinco filtros: o filtro de aquisição, o filtro de desmultiplexagem, o filtro de descompressão de áudio, o filtro de descompressão de vídeo, o filtro de áudio e o filtro de vídeo. O filtro de aquisição faz a aquisição dos dados áudio e vídeo de uma câmara IP. Como os dados de áudio e de vídeo utilizam o mesmo canal de comunicação, é necessário

fazer as desmultpiplexagem dos dados utilizando o filtro de desmultiplexagem. O filtro de descompressão de áudio e o filtro de descompressão de vídeo fazem a descompressão das amostras de áudio e de vídeo, respetivamente. Finalmente, o filtros de áudio e filtro de vídeo fazem a reprodução das amostras em função do respetivo tipo.

Tipo de Filtros O *DirectShow* define três tipos de filtros: os filtros de entrada, os filtros de transformação e os filtros de *rendering*.

Filtros de Entrada Os filtros de entrada representam a fonte de dados de áudio/vídeo para os restantes filtros do *filter graph*. Logo, todos os *filter graphs* de uma aplicação *DirectShow*, ou melhor, todas as *streams* de dados dos *filter graphs* de uma aplicação *DirectShow*, têm que ter pelo menos um filtro de entrada. Os exemplos mais comuns de filtros de entrada são os filtros de captura de áudio/vídeo e os filtros que permitem a reprodução de conteúdos multimédia. O *filter graph* da figura 4.2 é constituído por um filtro de entrada - o filtro de aquisição.

Filtros de Transformação Os filtros de transformação, tal como o próprio nome indica, transformam os dados recebidos à entrada do filtro. Isto é, os dados à saída do filtro são o resultado da aplicação de uma determinada função de transformação nos dados recebidos à entrada do filtro. Ao contrário dos filtros de entrada, os filtros de transformação são opcionais. Isto é, só é necessário utilizar um filtro de transformação se os dados à saída de um determinado filtro necessitarem de ser transformados de forma a que seja possível transmitir os dados a um outro filtro do mesmo *filter graph*. Os exemplos mais comuns de filtros de transformação são os filtros que implementam *codecs* e os filtros que implementam algoritmos de processamento de imagem. O *filter graph* da figura 4.2 é constituído por três filtros de transformação: o filtro de desmultiplexagem, o filtro de descompressão de áudio e o filtro de descompressão de vídeo.

Filtros de *Rendering* Os filtros de *rendering* são o destino dos dados de áudio/vídeo de qualquer *filter graph*. Logo, permitem analisar, consultar ou registar os dados à entrada do filtro utilizando algum mecanismo de *output*. Tal como acontece com os filtros de entrada, os *filter graphs* de uma aplicação *DirectShow* tem que ter pelo menos um filtro de saída. Os filtros que permitem a visualização dos

dados de vídeo e a reprodução dos dados de áudio são os exemplos mais comuns dos filtros de *rendering*. O *filter graph* da figura 4.2 é constituído por dois filtros de *rendering*: o filtro de áudio e o filtro de vídeo.

Pinos

Tal como foi referido anteriormente, são as ligações entre os vários filtros do *filter graph* que definem o fluxo de execução de dados da aplicação *DirectShow*. As ligações entre os vários filtros é possível através dos pinos, que são pontos de ligação específicos definidos em cada um dos filtros.

O *DirectShow* define dois tipos de pinos: os pinos de entrada e os pinos de saída. Os pinos de entrada recebem os dados de áudio/vídeo à entrada de um filtro. Os pinos de saída transmitem os dados de áudio/vídeo a um pino de entrada de um outro filtro do mesmo *filter graph*.

Para que a ligação entre dois pinos seja possível é necessário satisfazer os seguintes requisitos: (i) os pinos têm que ser de tipos diferentes, (ii) os pinos têm que pertencer a diferentes filtros do mesmo *filter graph*, (iii) os pinos têm que suportar o mesmo tipo de média, (iv) os pinos têm que chegar a um acordo de qual o mecanismo de transporte utilizado para a transmissão dos dados de áudio/vídeo e (v) um dos pinos tem que criar um objeto do tipo *Allocator* para criar e gerir os *buffers* de dados da *stream* que os pinos utilizam para a transmissão de dados de um pino para pino.

Se todas as condições apresentadas forem satisfeitas, a ligação entre os dois pinos é estabelecida com sucesso. Caso contrário, é apresentada um mensagem de erro que descreve o motivo de não ser possível a ligação entre os dois pinos.

Intelligent Connect O *Intelligent Connect* permite que dois filtros sejam ligados um ao outro de forma automática. Isto é, se for necessário estabelecer um ligação entre dois filtros e os dois filtros não tiverem dois pinos que sejam compatíveis, o *Intelligent Connect* faz o trabalho árduo de adicionar filtros de transformação intermédios ao *filter graph* de forma a que a ligação entre os dois filtros seja possível. Caso não seja encontrado um caminho de ligação entre os dois filtros recorrendo a filtros intermédios, é apresentada uma mensagem de erro a informar que a ligação entre os dois filtros não é possível.

4.3 Implementação do *DirectShow*

A *framework* multimédia *DirectShow* é constituída por um conjunto de classes desenvolvidas na linguagem de programação C++. O núcleo da *framework* é implementado na biblioteca dinâmica *quartz.dll*. Basicamente, a biblioteca *quartz.dll* é constituída pela classe que implementa o *filter graph*, as classes que implementam os filtros, as classes que implementam os pinos e as classes que servem de base de implementação das classes referidas anteriormente.

A base de implementação do *DirectShow* assenta na especificação COM (*Component Object Model*). Onde, todas as classes de objetos do *DirectShow* implementam interfaces, que definem um conjunto de regras que um componente tem que seguir de forma a que este possa ser utilizado num ambiente totalmente diferente daquele onde foi criado. Duas das principais vantagens em utilizar a especificação COM é o facto de se poder reutilizar os componentes sem ter em atenção a forma como eles são implementados internamente e o facto de ser possível utilizar o componente em linguagens de programação diferentes da linguagem de desenvolvimento nativa do componente (isto é, possibilita a programação multi-linguagem).

De seguida, é abordada a forma como cada um dos principais componentes do *DirectShow* é implementado, dando especial atenção às funcionalidades da linguagem de programação C++ que são utilizadas na sua implementação, e que prejudicam o desempenho das aplicações *DirectShow*, ao serem executadas em sistemas embebidos.

4.3.1 Interfaces e Múltipla Herança

Filter Graph

O *filter graph* é implementado no *DirectShow* na classe CFilterGraph. Como o SDK do *DirectShow* não disponibiliza o código fonte da classe CFilterGraph, foi necessário fazer a engenharia reversa do código binário da DLL *quartz.dll*, com o *disassembler* IDA Pro (Hex-Rays).

Para identificar as interfaces que são implementadas pela classe CFilterGraph e analisar se a classe utiliza múltipla herança, foi estudado o código *assembly* do construtor da classe. Como o código *assembly* do construtor da classe é demasiado extenso, apenas são apresentadas as linhas de código relacionadas com a implementação de

interfaces e a utilização da múltipla herança. Para uma melhor compreensão do código, em vez de ser apresentado o código completo do construtor da classe, é apresentado o código por blocos, seguindo-se a explicação de cada um deles.

```
mov esi, ecx ; esi contém o endereço de memória dos dados da classe CFilterGraph
...
lea edi, [esi+20h] ; edi contém o endereço de memória dos dados da classe CFilterGraph com o offset 0x20h
mov ecx, edi ; ecx contém endereço para dados serem inicializados pelo construtor da classe CServiceProvider
call CServiceProvider::CServiceProvider(void)
...
lea ebx, [esi+48h] ; ebx contém o endereço de memória dos dados da classe CFilterGraph com o offset 0x48h
mov ecx, ebx ; ecx contém endereço para dados serem inicializados pelo construtor da classe CBaseFilter
call CBaseFilter::CBaseFilter(ushort const *,IUnknown *,CCritSec *,_GUID const &)
```

Como o código do *DirectShow* foi compilado com o compilador *Microsoft Visual Studio C++*, por norma, o valor do apontador this da classe é guardado no registo ecx. Logo, a primeira linha de código *assembly* guarda no registo esi o endereço de memória dos dados da classe (o apontador this da classe).

De seguida, é chamado o construtor da classe CServiceProvider que inicializa os dados da classe CFilterGraph que estão na posição de memória com o *offset* 0x20h. A classe CServiceProvider implementa duas interfaces: a interface IServiceProvider e a interface IRegisterServiceProvider. A interface IServiceProvider define o método GetService, que devolve um objeto que implementa um serviço de um determinado tipo no *filter graph*. A interface IRegisterServiceProvider regista um objeto que implementa um serviço de um determinado tipo no *filter graph*. Como o *filter graph* define a interface IServiceProvider e a interface IRegisterServiceProvider e a classe CServiceProvider implementa as duas interfaces, implica que a classe CFilterGraph herda da classe CServiceProvider.

Por fim, é chamado o construtor da classe CBaseFilter que inicializa os dados da classe CFilterGraph que estão na posição de memória com o *offset* 0x48h. A implementação da classe CBaseFilter é explicada mais à frente quando for abordada a forma como os filtros são implementados. Tal como acontece com a classe CServiceProvider, a classe CFilterGraph também herda da classe CBaseFilter.

```
mov dword ptr [ebx], offset CFilterGraph::'vftable'{for 'CUnknown'}
xor ebx, ebx
lea ecx, [esi+9Ch]
mov dword ptr [esi], offset CFilterGraph::'vftable'{for 'IFilterGraph2'}
mov dword ptr [esi+4], offset CFilterGraph::'vftable'{for 'IGraphVersion'}
mov dword ptr [esi+8], offset CFilterGraph::'vftable'{for 'IPersistStream'}
mov dword ptr [esi+0Ch], offset CFilterGraph::'vftable'{for 'IObjectWithSite'}
mov dword ptr [esi+10h], offset CFilterGraph::'vftable'{for 'IAMMainThread'}
mov dword ptr [esi+14h], offset CFilterGraph::'vftable'{for 'IAMOpenProgress'}
mov dword ptr [esi+18h], offset CFilterGraph::'vftable'{for 'IAMGraphStreams'}
```

```
mov dword ptr [esi+1Ch], offset CFilterGraph::'vftable'{for 'IVideoFrameStep'}
mov dword ptr [edi], offset CFilterGraph::'vftable'{for 'IServiceProvider'}
mov dword ptr [esi+24h], offset CFilterGraph::'vftable'{for 'IRegisterServiceProvider'}
mov dword ptr [esi+54h], offset CFilterGraph::'vftable'{for 'IBaseFilter'}
mov dword ptr [esi+58h], offset CFilterGraph::'vftable'{for 'IAMovieSetup'}
```

O código *assembly* apresentado, inicializa os dados das tabelas virtuais das interfaces implementadas pela classe CFilterGraph. Assim, a classe CFilterGraph implementa, através da herança, treze interfaces. Como o número de interfaces implementadas pela classe é considerável, é explicada apenas a interface IFilterGraph2, visto ser a interface que desempenha um papel mais preponderante nas aplicações *DirectShow*.

A interface IFilterGraph2 herda das interfaces IFilterGraph e IGraphBuilder. Através desta interface, as aplicações *DirectShow* podem executar operações que permitem configurar o *filter graph*, tais como, adicionar filtros ao filter graph e estabelecer as ligações entre eles.

```
push 450h ; insere na stack o tamanho em bytes da classe CFGControl
call operator new(uint) ; aloca espaço em memória para a classe CFGControl
cmp eax, ebx ; testa se a memória foi alocada com sucesso
pop ecx ; remove da stack o parâmetro de entrada inserido
jz short loc_747E64CA ; em caso de erro, salta
push edi ; insere na stack a variável phr do tipo HRESULT*
push esi ; insere na stack o apontador this da classe CFilterGraph
mov ecx, eax ; ecx aponta para a memória alocada
call CFGControl::CFGControl(CFilterGraph *,long *)
```

Depois de inicializadas as tabelas virtuais de cada interface no construtor da classe, é alocado espaço em memória para um objeto do tipo CFGControl. Se os dados forem alocados com sucesso, é chamado o construtor da classe CFGControl para inicializar os dados. Basicamente, a classe CFGControl encapsula todas as funcionalidades de controlo e registo de eventos da classe CFilterGraph, tais como, iniciar, fazer pausa, parar o fluxo de execução de dados do *filter graph* e definir as funções *callback* para o registo de eventos, através das interfaces IMediaControl e IMediaEventEx.

A figura 4.3 apresenta um diagrama onde é possível visualizar a forma como os dados da classe CFilterGraph estão organizados em memória, as interfaces que a classe implementa e os métodos que cada uma das interfaces define.

Ao analisar o diagrama, conclui-se que a classe CFilterGraph implementa treze interfaces e utiliza múltipla herança, ao herdar das classes CServiceProvider e da classe CBaseFilter.

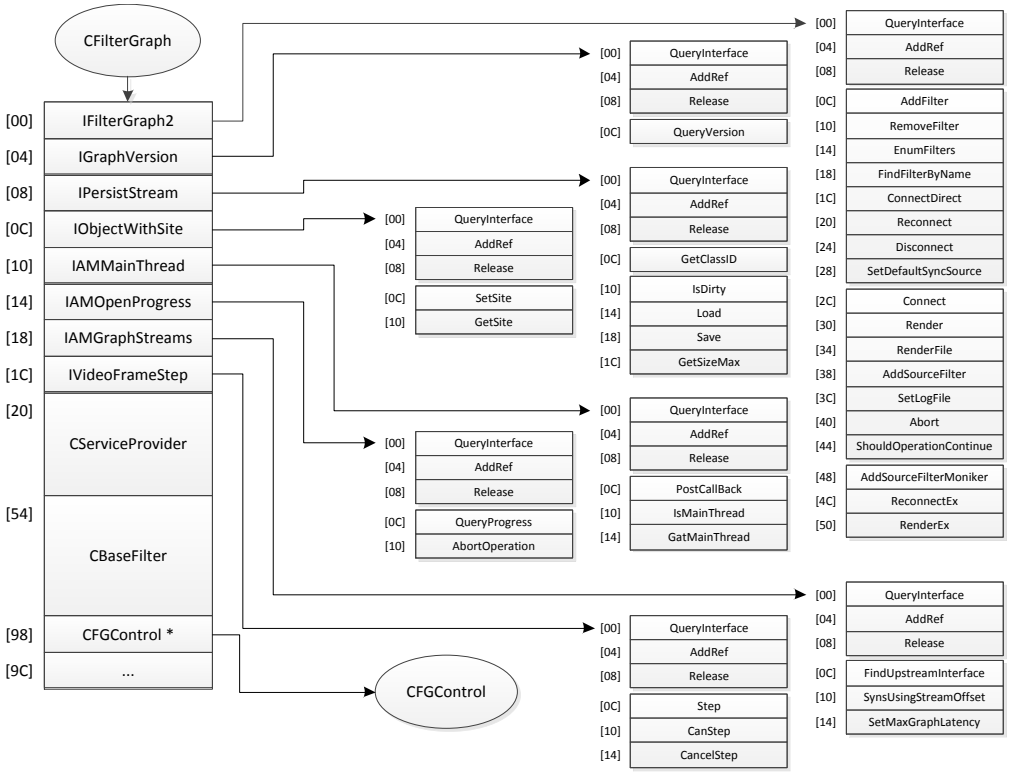


Figura 4.3: Organização dos dados da classe CFilterGraph em memória.

Filtro

Ao contrário do *filter graph*, o código fonte das classes que servem de base de implementação de um filtro, é disponibilizado no SDK do *DirectShow*. A razão de o código fonte das classes ser disponibilizado, prende-se com o facto de a *framework* ser extensível e ser possível implementar novos filtros para adicionar novas funcionalidades à *framework*.

Classe Abstrata *CBaseFilter* Todos os filtros que são implementados no *DirectShow* herdam da classe abstrata CBaseFilter. A classe abstrata CBaseFilter serve de base de implementação para todos os tipos de filtros, sejam eles, filtros de entrada, filtros de transformação ou filtros de *rendering*. A figura 4.4, apresenta o diagrama de classes da classe abstrata CBaseFilter:

A classe abstrata CBaseFilter implementa duas interfaces: a interface IBaseFilter e a interface IAMovieSetup. A interface IBaseFilter é utilizada pelo *filter graph* e pelas aplicações *DirectShow* para comunicar com os filtros. Dos métodos que a interface define, destacam-se dois: o método EnumPins e o método FindPin. O método EnumPins enumera os pinos do filtro. O método FindPin devolve um apontador para o pino

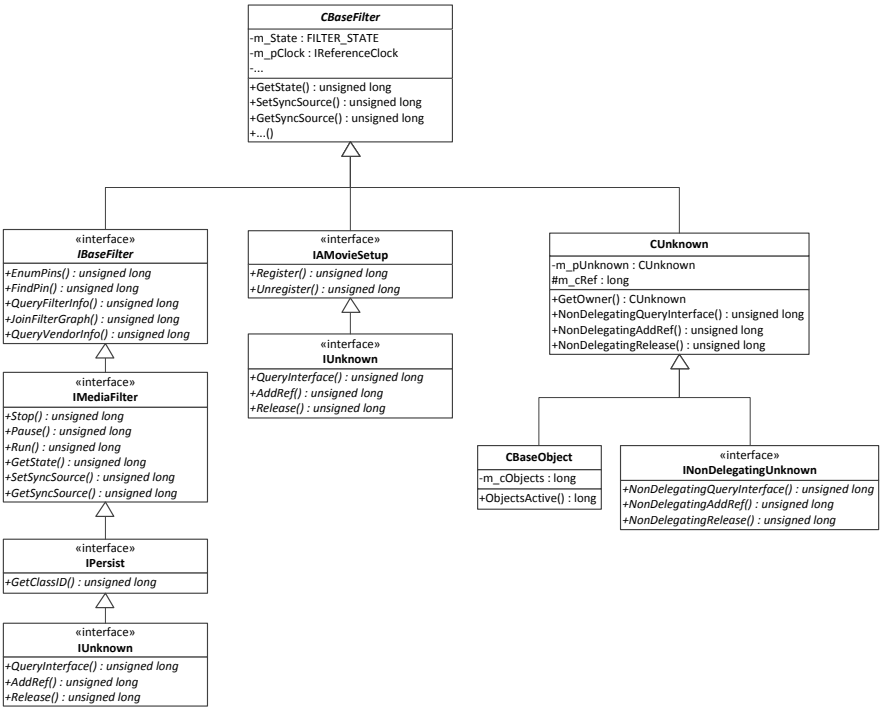


Figura 4.4: Diagrama de classes da classe CBaseFilter.

do filtro com índice n.

Como a interface IBaseFilter herda da interface IMediaFilter, também define métodos que permitem controlar o estado do fluxo de execução de dados do filtro. Assim, é possível saber o estado do fluxo de execução de dados do filtro através do método GetState. Para iniciar, fazer pausa ou parar o fluxo de execução de dados, são definidos os métodos Run, Pause e Stop, respetivamente.

A interface IAMovieSetup define métodos que permitem que objetos implementados em bibliotecas dinâmicas (DLLs), possam ser registados de forma automática na *framework*. Assim, o método Register permite registar o filtro na lista de filtros activos da *framework* e o método Unregister permite remover o filtro da lista.

A classe abstrata CBaseFilter também herda da classe CUnknown. A classe CUnknown permite criar objetos COM, que podem ser combinados com outros objetos COM para suportar múltiplas interfaces. Assim, a classe CUnknown implementa a interface IUnknown que permite obter apontadores das interfaces implementadas pelo filtro (método QueryInterface) e gerir o tempo de vida do filtro através dos métodos AddRef e Release.

Tipos de Filtros Embora a classe abstrata CBaseFilter sirva de base de implementação de todos os filtros do *DirectShow*, o *DirectShow* implementa uma classe base para cada um dos tipos de filtros. A figura 4.5 apresenta o diagrama de classes das classes que servem de base de implementação dos filtros.

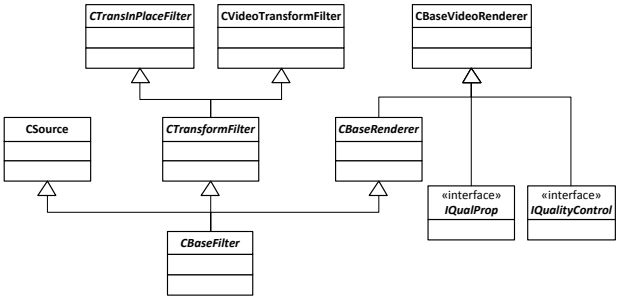


Figura 4.5: Diagrama de classes das classes que servem de base de implementação dos filtros.

Ao analisar o diagrama de classes da figura 4.5, verifica-se que a classe abstrata CBaseFilter serve de base de implementação de três classes: a classe CSource, a classe CTransformFilter e a classe CBaseRenderer. A classe CSource é a classe base utilizada para implementar os filtros de entrada. A classe abstrata CTransformFilter serve de base de implementação dos filtros de transformação. Por fim, a classe abstrata CBaseRenderer é utilizada como base de implementação dos filtros de *rendering*.

Ao implementar um filtro de transformação é necessário ter em consideração os seguintes aspectos: (i) saber se o filtro de transformação implementa um *codec* e (ii) de que forma são geridos os dados de entrada e de saída do filtro de transformação. Ao desenvolver um filtro de transformação que implemente um *codec*, em vez de se utilizar a classe abstrata CTransformFilter como base de implementação, é utilizada a classe abstrata CVideoTransformFilter. A classe abstrata CVideoTransformFilter adiciona à classe base CTransformFilter mecanismos de controlo de qualidade que são utilizados pelos *codecs*. No que diz respeito à forma como são geridos os dados de entrada e de saída de um filtro de transformação, o *DirectShow* disponibiliza duas classes: a classe CTransformFilter e a classe CTransInPlaceFilter. A classe CTransformFilter utiliza *buffers* de dados diferentes para os dados de entrada e para os dados de saída do filtro, isto é, os dados de entrada não são alterados ao executar a operação de transformação pelos dados de saída. A classe CTransInPlaceFilter utiliza o mesmo buffer de dados para os dados de entrada e para os dados de saída do filtro.

Tal como foi referido anteriormente, a classe abstrata CBaseRenderer é utilizada como base de implementação dos filtros de *rendering*. Caso o filtro de *rendering* a desen-

volver necessite de implementar mecanismos de visualização de imagens de vídeo, em vez de se utilizar a classe CBaseRenderer como base de implementação do filtro, é utilizada a classe abstrata CBaseVideoRenderer. A classe abstrata CBaseVideoRenderer herda da classe abstrata CBaseRenderer e implementa as interfaces IQualProp e a interface IQualityControl. A interface IQualProp define métodos que permitem obter informação sobre o desempenho do *rendering* de vídeo. A interface IQualityControl define mecanismos que suportam o controlo de qualidade, tais como, gerar e receber mensagens de controlo de qualidade.

Pino

Tal como já foi referido, os pinos são os pontos de ligação que permitem estabelecer as ligações entre os filtros do mesmo *filter graph*. Tal como o SDK do *DirectShow* disponibiliza o código fonte das classes que servem de base de implementação de um filtro, também é disponibilizado o código fonte das classes que servem de base de implementação de um pino. Isso deve-se ao facto de na implementação de um novo filtro, para dar suporte a uma nova funcionalidade na *framework*, ser também necessário implementar novos pinos para o filtro.

Classe Abstrata *CBasePin* Todos os pinos do *DirectShow* herdam da classe abstrata CBasePin. A figura 4.6, apresenta o diagrama de classes da classe abstrata CBasePin.

A classe abstrata CBasePin implementa duas interfaces: a interface IPin e a interface IQualityControl. A interface IPin define métodos que permitem saber informações sobre o estado do pino (ligado ou desligado), os tipos de média suportados pelo pino e as operações de controlo que permitem gerir a ligação com o pino. A interface IPin assume um papel preponderante no desenvolvimento de aplicações *DirectShow*, visto ser a interface utilizada para gerir as ligações entre os pinos de filtros do mesmo *filter graph*.

A interface IQualityControl define métodos que fornecem o suporte ao controlo de qualidade do fluxo de execução de dados de áudio/vídeo do pino. Os pinos ao implementar esta interface, podem enviar e receber mensagens de controlo de qualidade através dos métodos Notify e SetSink, respetivamente.

Como a classe CBasePin implementa um objeto COM, está implícito que a classe herda da classe CUnknown.

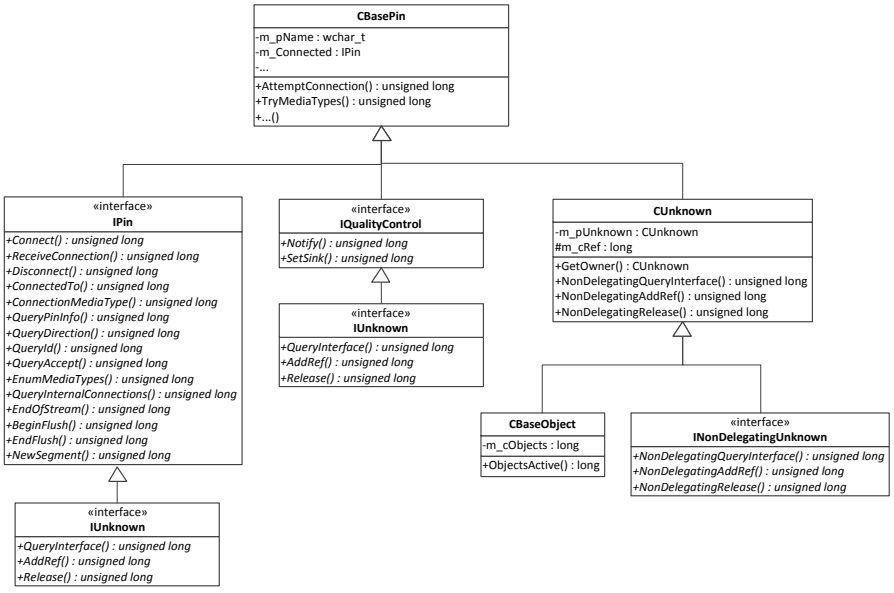


Figura 4.6: Diagrama de classes da classe CBasePin.

Tipos de Pinos Tal como já foi referido, a classe abstrata CBasePin serve de base de implementação de todo o tipo de pinos do *DirectShow*, sejam eles pinos de entrada ou pinos de saída. Embora a classe possa ser utilizada de forma direta para implementar um pino, o *DirectShow* disponibiliza outras classes que se adequam mais na implementação de pinos com determinadas características. A figura 4.7 apresenta o diagrama de classes das classes que servem de base de implementação dos pinos.

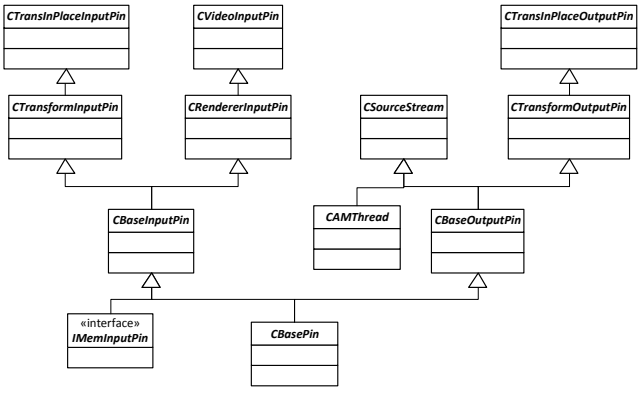


Figura 4.7: Diagrama de classes das classes que servem de base de implementação dos pinos.

A classes abstratas CBaseInputPin e CBaseOutputPin herdam da classe abstrata CBase-Pin. A classe abstrata CBaseInputPin implementa as funcionalidades básicas de um pino de entrada. Para tal, implementa a interface IMemInputPin que define métodos

que permitem negociar com um pino de saída a ligação e receber de um pino de saída os dados de áudio/vídeo. A classe abstrata CBaseOutputPin implementa as funcionalidades básicas de um pino de saída. Para que um pino de saída possa enviar a um pino de entrada os dados de áudio/vídeo, é definida uma variável apontador do tipo IMemInputPin na classe CBaseOutputPin que aponta para o pino de entrada ao qual está ligado (CBaseInputPin).

Dependendo do tipo de filtro a implementar, o *DirectShow* define classes que servem de base de implementação dos pinos que compõem cada um dos tipos de filtros. Assim, é definida a classe CSourceStream que implementa um pino de saída de um filtro de entrada. A classe CSourceStream herda da classe CBaseOutputPin e da classe CAMThread. A classe CAMThread implementa um *worker thread* que faz a aquisição de dados de áudio/vídeo de uma fonte de dados. Se o filtro a implementar for um filtro de transformação o *DirectShow* define quatro classes que servem de base de implementação dos pinos. Se for utilizada a classe CTransformFilter como base de implementação do filtro de transformação, são utilizadas as classes CTransformInputPin e CTransformOutputPin para implementar o pino de entrada e o pino de saída do filtro, respetivamente. Caso seja utilizada a classe CTransInPlaceFilter para implementar o filtro de transformação, são utilizadas as classes CTransInPlaceInputPin e CTransInPlaceOutputPin para implementar os pinos do filtro. Finalmente, se for implementado um filtro de *rendering* utilizando a classe base CBaseRenderer, é usada a classe CRendererInputPin para implementar o pino de entrada do filtro de *rendering*. Caso seja implementado um filtro de *rendering* com suporte à visualização de vídeo (classe base CBaseVideoRenderer) é utilizada a classe base CVideoInputPin para implementar o pino de entrada do filtro.

4.3.2 Criação Dinâmica dos Objetos

Numa aplicação *DirectShow*, as instâncias dos componentes do *DirectShow* que implementam a especificação COM, são criadas de forma dinâmica através da API do Windows CoCreateInstance. Os componentes que não implementam a especificação COM são criados de forma dinâmica utilizando a função new.

Filter Graph

Para criar um *filter graph* numa aplicação *DirectShow*, é utilizada a API CoCreateInstance com os seguintes parâmetros de entrada: o identificador de classe CL-

SID_FilterGraph, a referência do identificador da interface IID_IGraphBuilder e a variável apontador pGraph que é do tipo IGraphBuilder.

```
IGraphBuilder *pGraph = NULL; // apontador para a interface IGraphBuilder do filter graph
IMediaControl *pControl = NULL; // apontador para a interface IMediaControl do filter graph
IMediaEvent *pEvent = NULL; // apontador para a interface IMediaEvent do filter graph
...
// Cria uma instância do filter graph, obtendo a interface IGraphBuilder
hr = CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
                      IID_IGraphBuilder, (void **)&pGraph);
...
// obtém a interface IMediaControl do filter graph
hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);
...
// obtém a interface IMediaEvent do filter graph
hr = pGraph->QueryInterface(IID_IMediaEvent, (void **)&pEvent);
...
```

Para comunicar com o *filter graph*, são utilizadas essencialmente três interfaces: a interface IGraphBuilder, a interface IMediaControl e a interface IMediaEvent. A interface IGraphBuilder define métodos que permitem adicionar/remover filtros e estabelecer as ligações entre os filtros. A interface IMediaControl permite controlar o fluxo de execução de dados do filter graph, tais como, iniciar, fazer pausa e parar o fluxo de execução de dados. Finalmente, a interface IMediaEvent permite registar as *callbacks* para o processamento de eventos e enviar eventos para o *filter graph*.

Filtro

Para adicionar os filtros ao *filter graph* é necessário, numa primeira fase, alocar em memória uma instância de cada um dos filtros através da API do Windows CoCreateInstance. As linhas de código seguintes permitem criar em memória três filtros: o filtro de entrada com o identificador de classe CLSID_PushSource, o filtro de transformação com o identificador de classe CLSID_ColorSpaceConverter e o filtro de *rendering* com o identificador de classe CLSID_VideoRenderer.

```
IBaseFilter *pSourceFilter = NULL;
IBaseFilter *pConverterFilter = NULL;
IBaseFilter *pRendererFilter = NULL;
...
// cria uma instância do filtro de entrada com o identificador CLSID_PushSource
hr = CoCreateInstance(CLSID_PushSource, NULL, CLSCTX_INPROC_SERVER,
                      IID_IBaseFilter, (void **)&pSourceFilter);
...
// cria uma instância do filtro de transformação com o identificador CLSID_ColorSpaceConverter
hr = CoCreateInstance(CLSID_ColorSpaceConverter, NULL, CLSCTX_INPROC_SERVER,
                      IID_IBaseFilter, (void **)&pConverterFilter);
...
```

```
// cria uma instância do filtro de rendering com o identificador CLSID_VideoRenderer
hr = CoCreateInstance(CLSID_VideoRenderer, NULL, CLSCTX_INPROC_SERVER,
                      IID_IBaseFilter, (void **)&pRendererFilter);
...
```

Depois de alocar em memória uma instância de cada um dos filtros que compõem o *filter graph*, é necessário adicionar cada um dos filtros ao *filter graph* utilizando o método AddFilter da interface IGraphBuilder implementada pela classe CFilterGraph.

```
// Adicionar o filtro Push Source ao filter graph
hr = pGraph->AddFilter(pSourceFilter, L"Generic Source Filter");
...
// Adicionar o filtro Color Space Converter ao filter graph
hr = pGraph->AddFilter(pConverterFilter, L"Color Space Converter");
...
// Adiconar o filtro Video Rendering ao filter graph
hr = pGraph->AddFilter(pRendererFilter, L"Video Renderer");
...
```

Os filtros adicionados ao *filter graph* são guardados numa lista ligada. Sempre que é necessário executar uma determinada operação nos filtros do *filter graph*, é percorrida a lista ligada de filtros e executada a operação pretendida em cada um deles.

Pinos

Os pinos também são criados de forma dinâmica dentro dos filtros. Ao analisar o código que define as classes que servem de base de implementação dos filtros, as variáveis da classe que implementam os pinos são todas do tipo apontador. Para alocar espaço em memória para cada um dos pinos é utilizado o operador new.

```
// definição da classe CTransformFilter
class CTransformFilter : public CBaseFilter {
public:
    ...
protected:
    ...
    CTransformInputPin *m_pInput; // pino de entrada do filtro
    CTransformOutputPin *m_pOutput; // pino de saída do filtro
};

// método GetPin da classe CTransformFilter
CBasePin* CTransformFilter::GetPin(int n) {
    ...
    // cria o pino de entrada do filtro
    m_pInput = new CTransformInputPin(NAME("Transform input pin"),
                                      this, &hr, L"XForm In");
    ...
    // cria o pino de saída do filtro
```

```
        m_pOutput = new CTransformOutputPin(NAME("Transform output pin"),
                                    this, &hr, L"XForm Out");
        ...
}
```

Para ligar dois pinos de filtros do mesmo *filter graph* é utilizado o método Connect da interface IGraphBuilder do *filter graph*.

```
// obtém o pino de saída do filtro Push Source
IPin *pSourceOut = GetPin(pSourceFilter, PINDIR_OUTPUT);
if (pSourceOut != NULL) {
        // obtém o pino de entrada do filtro Color Space Converter
        IPin *pConvIn = GetPin(pConverterFilter, PINDIR_INPUT);
                if (pConvIn != NULL) {
                // liga o pino de saída ao pino de entrada
                        hr = pGraph->Connect(pSourceOut, pConvIn);
                }
    }
```

O *filter graph* inicializa o processo de ligação entre os dois pinos através do método Connect. O método Connect do *filter graph* chama o método Connect da interface IPin do filtro de saída (pSourceOut), passando como parâmetro de entrada o pino de entrada (pConvIn). Se os dois pinos forem compatíveis, e estabelecida a ligação entre os dois filtros.

4.3.3 Tratamento de Exceções

Na plataforma x86 da Intel, o registo FS aponta para o valor atual da estrutura do *Thread Information Block* (TIB). Um dos elementos da estrutura TIB é um apontador para uma estrutura do tipo EXCEPTION_RECORD, que contém um apontador para a função *callback* responsável pelo tratamento da exceção. Assim, é reservado na *stack* o espaço de memória necessário para as estruturas de tratamento de exceções à medida que as funções vão sendo processadas. As estruturas EXCEPTION_RECORD formam uma lista ligada, isto é, a nova estrutura EXCEPTION_RECORD contém um apontador para a estrutura EXCEPTION_RECORD anterior, e assim sucessivamente. Logo, o início da lista é sempre apontado pelos primeiros quatro *bytes* do TIB indexadas no segmento FS do x86, isto é FS:[0].

Muitos filtros do *DirectShow* utilizam exceções para o tratamento de erros. As exceções foram identificadas no código binário da biblioteca dinâmica quartz.dll quando o registo FS é utilizado. Exemplo disso, é o filtro de transformação CMjpegDec que implementa o algoritmo de descompressão de imagens no formato JPEG. O mé-

todo Transform do filtro, que é utilizado para fazer a descompressão das *frames* contém as seguintes linhas de código *assembly* :

```
?Transform@CMjpegDec@@UAEJPAUIMediaSample@@0@Z proc near
    push 0ACh
    push offset stru_74821EF0
    call ___SEH_prolog
    ...
    call ___SEH_epilog
    retn 8
?Transform@CMjpegDec@@UAEJPAUIMediaSample@@0@Z
```

A função *assembly* ___SEH_prolog inicializa os primeiros quatro *bytes* do registo FS com uma nova estrutura EXCEPTION_RECORD para o tratamento de uma nova exceção:

```
___SEH_prolog proc near
    push offset ___except_handler3
    mov eax, large fs:0
    push eax
    mov eax, [esp+8+arg_4]
    ...
    mov large fs:0, eax
    retn
___SEH_prolog endp
```

A função *assembly* ___SEH_epilog remove a estrutura EXCEPTION_RECORD apontada por FS:[0], fazendo com que FS:[0] aponte para a estrutura EXCEPTION_RECORD anterior na lista ligada de exceções.

```
___SEH_epilog proc near
    mov ecx, [ebp-10h]
    mov large fs:0, ecx
    ...
    retn
___SEH_epilog endp
```

4.4 Impacto do *DirectShow* em Sistemas Embebidos

Na secção anterior, foi apresentada a forma como os principais componentes do *DirectShow* são implementados (*filter graphs*, filtros e pinos). Depois de analisar o código que implementa cada um dos componentes, chegou-se à conclusão que são utilizadas técnicas de programação dinâmicas da linguagem de programação C++ que produzem um impacto negativo no desempenho e no tamanho do ficheiro

executável das aplicações *DirectShow*, especialmente importante em sistemas embebidos.

Das funcionalidades enumeradas, a que produz maior impacto no desempenho e no tamanho do ficheiro executável das aplicações *DirectShow* é o uso intensivo de interfaces e funções virtuais. Esta funcionalidade introduz indireções no código da aplicação, que afetam a linearidade do código e consequentemente o desempenho da aplicação. A linearidade do código é afetada porque, ao compilar o código da aplicação não é possível saber o fluxo de execução da mesma. No ato da compilação, o compilador não sabe o fluxo de execução de uma função virtual, porque o número de caminhos possíveis depende do número de classes que reimplementam a função virtual, sendo este apenas resolvido em tempo de execução. Para não falar do facto de todas as funções virtuais de uma classe terem que ser resolvidas pelo *linker* no ato da compilação da aplicação. Desta forma, aumenta a probabilidade de existirem funções que são compiladas e que nunca serão executadas, poluindo assim o código executável da aplicação.

Para além disso, todos os objetos são criados dinamicamente (especialmente os filtros). Como as instâncias de cada um dos filtros são criadas de forma dinâmica em tempo de execução, as ligações entre eles também são estabelecidas em tempo de execução. Assim, não é possível aproveitar as sinergias que possam existir entre os vários filtros do mesmo *filter graph*. Quando é estabelecida uma ligação entre dois filtros em tempo de execução, não é possível remover o código de cada um dos filtros que implementa funcionalidades variáveis que não serão utilizadas.

Para exemplificar o que foi dito, é analisado o código *assembly* do filtro de transformação CMjpegDec, que está implementado na biblioteca do *DirectShow quartz.dll*.

4.4.1 Filtro de transformação *CMjpegDec*

O filtro de transformação CMjpegDec implementa o *decoder* JPEG que permite fazer a descompressão de imagens no formato JPEG para um formato descomprimido (*raw*). As linhas de código seguintes apresentam partes do código *assembly* do construtor da classe CMjpegDec:

```
??0CMjpegDec@@QAE@PAGPAUIUnknown@@PAJ@Z proc near
    ...
    push offset _CLSID_MjpegDec ; identificador da classe (REFCLSID clsid)
    push [ebp+arg_4] ; o owner da classe (LPUNKNOWN pUnk)
    mov esi, ecx ; esi aponta para os dados da classe (this)
    push [ebp+arg_0] ; nome do filtro (TCHAR *pName)
```

```
        call CVideoTransformFilter::CVideoTransformFilter(...) ; chama o construtor da classe base
        ...
        mov dword ptr [esi], offset CMjpegDec::'vftable'{for 'CUnknown'}
        mov dword ptr [esi+0Ch], offset CMjpegDec::'vftable'{for 'IBaseFilter'}
        mov dword ptr [esi+10h], offset CMpegAudioCodec::'vftable'{for 'IAMovieSetup'}
        ...
??0CMjpegDec@@QAE@PAGPAUIUnknown@@PAJ@Z endp
```

Ao analisar o construtor da classe, chega-se à conclusão que a classe CMjpegDec herda da classe abstrata CVideoTransformFilter, que é a classe base do *DirectShow* que permite criar filtros de transformação que implementam *codecs*. As linhas de código seguintes apresentam partes do código *assembly* do método Transform da classe CMjpegDec:

```
?Transform@CMjpegDec@@UAEJPAUIMediaSample@@0@Z proc near
        ...
        push [ebp+pInfoHeader] ; cabeçalho da imagem
        push edi ; aponta para o buffer com a imagem a descomprimir
        push dword ptr [esi+0C0h] ; aponta para o buffer da imagem destino
        call DecompressBegin(...) ; descomprime a imagem
        ...
?Transform@CMjpegDec@@UAEJPAUIMediaSample@@0@Z endp
```

O método Transform recebe duas amostras de média do tipo IMediaSample*: a amostra pIn aponta para a imagem a descomprimir e a amostra pOut aponta para a imagem descomprimida. A função DecompressBegin faz a descompressão da imagem JPEG para o formato descomprimido pretendido. As linhas de código seguintes apresentam partes do código *assembly* da função DecompressBegin:

```
_Decompress@12 proc near
        ...
        lea eax, [esi+8] ; lê o endereço de memória da variável de erros
        push eax ; passa o apontador com a informação das callbacks para processamento de erros
        call jpeg_exception_error(...)
        mov [edi], eax ; contém a informação do contexto do decoder
        push edi ; passa o apontador do contexto do decoder
        call jpeg_create_decompress(...) ; faz a descompressão da imagem JPEG
        ...
_Decompress@12 endp
```

A função DecompressBegin utiliza duas funções: a função jpeg_exception_error e a função jpeg_create_decompress. As duas funções pertencem à biblioteca IJG JPEG que foi desenvolvida pelo *Independent JPEG Group*. Logo, o filtro CMjpegDec utiliza a biblioteca IJG JPEG no processo de descompressão da imagem JPEG.

Ao analisar cada um das funções, verifica-se que a biblioteca foi compilada com todas as funcionalidades. Desta forma, conclui-se que independentemente da configuração do *filter graph* onde este filtro é inserido, o filtro CMjpegDec contém sempre

o mesmo código não sendo possível retirar o código das funcionalidades que não são utilizadas. Como resultado, o filtro implementa todas as variações do formato JPEG fazendo com que a maior parte do código do filtro não seja executado e polua o ficheiro executável da aplicação.

4.5 Conclusão

Neste capítulo foram apresentadas as funcionalidades da linguagem de programação C++ que são utilizadas para implementar os principais componentes do *DirectShow* (*filter graphs*, filtros e pinos). Das funcionalidades utilizadas, deu-se especial destaque àquelas que prejudicam o desempenho e o tamanho do ficheiro executável das aplicações *DirectShow*, especialmente quando estas são compiladas para serem executadas em sistemas embebidos.

Da análise do código *assembly* resultante da engenharia reversa da biblioteca do Windows quartz.dll e do código C++ do SDK do *DirectShow*, conclui-se que os componentes do *DirectShow* utilizam interfaces, funções virtuais e exceções. Embora alguns componentes utilizem exceções para o tratamento de erros, o uso excessivo de interfaces e funções virtuais é que influencia de forma significativa o desempenho e o tamanho do ficheiro executável das aplicações *DirectShow*.

Ao utilizar interfaces e funções virtuais, são introduzidas indirecções no código, que por sua vez fazem com que o código da aplicação não seja linear. Assim, aumenta a probabilidade da ocorrência de *cache misses* e *page faults*, que penalizam, em muito, o desempenho das aplicações *DirectShow*. Outro problema relacionado com a utilização de funções virtuais, prende-se com o facto de estas serem resolvidas apenas em tempo de execução. Logo, não é possível saber em tempo de compilação as funcionalidades variáveis que cada um dos filtros vai utilizar, fazendo com que não seja possível remover o código que não é executado.

Como os filtros de uma aplicação *DirectShow* são criados de forma dinâmica e a ligação entre eles só é estabelecida em tempo de execução, não é possível aproveitar as sinergias que possam existir entre os vários filtros do mesmo *filter graph*, reduzindo assim o tamanho do ficheiro executável da aplicação *DirectShow*.

Assim, conclui-se que a forma como os componentes do *DirectShow* são implementados penaliza significativamente o desempenho e o tamanho do ficheiro executável das aplicações *DirectShow* quando estas são executadas em sistemas embebidos.

Capítulo 5

Frameworks TMP e a Gestão da Variabilidade

5.1 Introdução

Nos capítulos anteriores, foram apresentadas as principais técnicas da linguagem de programação C++ que permitem gerir a variabilidade do código de sistemas complexos. Das técnicas apresentadas, o polimorfismo dinâmico é, sem duvida, a mais utilizada pela indústria para gerir a variabilidade do código. Embora o polimorfismo dinâmico apresente vantagens ao nível da facilidade de implementação, produz um impacto negativo no desempenho e no tamanho do ficheiro executável do sistema, especialmente quando este é compilado para ser executado em sistemas embebidos. A *framework* multimédia *DirectShow* é um exemplo claro que ilustra bem o que foi referido anteriormente.

Neste capítulo, é apresentado o *template metaprogramming* (TMP) como a principal técnica da linguagem de programação C++ para resolver a variabilidade do código de um sistema em tempo de compilação. Assim, numa primeira fase é feita uma pequena introdução ao TMP e dos blocos básicos necessários para desenvolver código em TMP. Por fim, são apresentadas as principais *frameworks* que utilizam TMP para gerir a variabilidade, dando especial atenção à biblioteca boost.MPL e à biblioteca da *Adobe* GIL (*Generic Image Library*).

5.2 *Template Metaprogramming*

Template metaprogramming (TMP) é uma técnica de metaprogramação que utiliza *templates* para gerar e manipular o código de uma aplicação em tempo de compilação (Czarnecki and Eisenecker, 2000; Abrahams and Gurtovoy, 2004; Gennaro, 2011). Ao desenvolver código utilizando TMP, é possível extender as capacidades do compilador C++, fazendo com que este aja como um interpretador, que emite todas as instruções necessárias para produzir uma configuração estática e optimizada de uma aplicação. Depois do código TMP ser instanciado pelo compilador C++, é produzido código optimizado tal como se este fosse escrito à mão.

Como o código TMP, depois de compilado, traduz-se em código C++ com um nível de otimização elevado, é a técnica ideal para gerir a variabilidade do código de aplicações complexas em tempo de compilação. Assim, é possível criar vários artefactos de uma aplicação utilizando *templates* e TMP, e de acordo com os requisitos do cliente, compilar apenas o código necessário para a configuração pretendida.

Embora o código TMP se traduza em código C++ totalmente otimizado, a grande desvantagem de utilizar TMP, prende-se com o facto de os programadores necessitarem de dominar tópicos avançados da linguagem de programação C++ (curva de aprendizagem é normalmente lenta o que inicialmente pode afetar a produtividade), tais como, as *templates*. Ao desenvolver código TMP, o programador precisa de aprender a pensar de uma forma diferente, porque o código TMP está mais relacionado com a programação funcional do que com a programação imperativa. Assim, a sintaxe e os idiomas do TMP são isotéricos comparados com a programação convencional em C++, sendo muito mais difíceis de compreender (Cardoso et al., 2012g).

5.2.1 Blocos Básicos do TMP

Tal como referido anteriormente, o código TMP é implementado de forma diferente do código normalmente implementado na linguagem de programação C++. A grande diferença entre as duas implementações, reside no facto de o código C++ ser imperativo e o código TMP ser funcional (Cardoso et al., 2012g).

Como o código TMP é funcional, implica que não existem variáveis, declarações de atribuição, construtores condicionais e construtores iterativos ou cíclicos. O código TMP baseia-se, essencialmente, em conceitos de funções matemáticas, em que

cada processo iterativo está dividido em vários sub-processos, onde normalmente se utilizam funções recursivas (Cardoso et al., 2012g).

As próximas subseções explicam cada um dos blocos básicos que compõem o código C++ utilizando *template metaprogramming*.

"Variáveis"

No código TMP o valor das "variáveis" não podem ser alterados porque são definidas utilizando typedefs ou constantes inteiras através de enumerações.

O código seguinte apresenta a definição de um tipo de dados (*named values*) e de uma constante inteira (*integer values*):

```
// definição de um tipo de dados (named value)
struct NamedValue {
    typedef int value;
};

// definição de uma constante (integer value)
struct IntegerValue {
    enum { value = 2 };
};
```

Para utilizar o tipo de dados value da estrutura de dados NamedValue, é necessária a seguinte linha de código:

```
NamedValue::value age = 35;
```

O código define a variável age do tipo int e inicializa a variável a 35. Para utilizar a constante value definida na estrutura de dados IntegerValue, é necessária a seguinte linha de código:

```
int number = IntegerValue::value;
```

O código define a variável number do tipo int e inicializa a variável com o valor da constante value definida na estrutura de dados IntegerValue.

Funções

As funções (mais precisamente, metafunções) são definidas em TMP utilizando estruturas *template* ou classes *template*. Para passar argumentos para às metafunções, são utilizados os parâmetros de entrada da *template*. Para retornar valores ou tipos de dados são utilizadas constantes (*integer values*) ou typedefs (*named values*).

O código seguinte apresenta um exemplo da metafunção Add:

```
// definição da metafunção
template<int X, int Y>
struct Add {
        // define o tipo de dados do resultado
        typedef int result_type;
        // define o valor do resultado
        enum { result = X + Y } ; // store the result value
};
```

O código seguinte define o valor da variável result com o resultado da metafunção Add quando os parâmetros de entrada são 2 e 3:

```
int result = Add<2, 3>::result;
```

Como a constante result da estrutura de dados Add é resolvido em tempo de compilação, a variável result é inicializada a 5.

Salto Condicional

Quando é necessário utilizar saltos condicionais são utilizadas *templates* especializadas. Ao utilizar *templates* especializadas, no ato da compilação o compilador decide qual a *template* especializada que mais se adequa em função dos parâmetros de entrada da *template*.

O código seguinte apresenta um exemplo que verifica se dois tipos de dados são do mesmo tipo:

```
// definição da template genérica
template<typename T, typename U>
struct is_same {
        enum { result = 0 };
};

// definição da template especializada
template<typename T>
struct is_same<T, T> {
        enum { result = 1 };
};
```

A metafunção is_same recebe dois parâmetros de entrada de *template*: o tipo de dados T e o tipo de dados U. Para implementar a metafunção são definidas duas *templates*: uma *template* genérica e uma *template* especializada. A *template* genérica é executada sempre que os tipos de dados T e U são diferentes, sendo devolvido o resultado 0 (falso) através da constante result. Por sua vez, a *template* especiali-

zada é executada sempre que os tipos de dados T e U são iguais, sendo devolvido o resultado 1 (verdadeiro) através da constante result.

O código seguinte apresenta um exemplo da utilização da metafunção is_same:

```
bool result = is_same<int, char>::result;
```

No código apresentado, é utilizada a metafunção is_same para verificar se os parâmetros de entrada da *template* int e char são do mesmo tipo. Como os parâmetros de entrada da *template* não são do mesmo tipo, é chamada a *template* genérica da metafunção is_same e a variável result é inicializada a falso.

Recursividade

Tal como acontece com as linguagens funcionais, o código TMP utiliza a recursividade em vez de utilizar construtores cíclicos. A condição de paragem de uma função recursiva é definida através de *templates* especializadas.

O código seguinte apresenta a metafunção factorial:

```
// definição da template genérica
template <unsigned n>
struct factorial {
        enum { value = n * factorial<n - 1>::value };
};

// definição da condição de paragem
template <>
struct factorial<0> {
        enum { value = 1 };
};
```

A metafunção factorial recebe como parâmetros de entrada de *template* a variável n, para calcular o factorial de n. Para implementar a metafunção factorial, são definidas duas *templates*: uma *template* genérica e uma *template* especializada. A *template* genérica chama de forma recursiva a mesma *template* até chegar à condição de paragem da recursividade. Para implementar a condição de paragem é definida a *template* especializada quando o valor do parâmetro de entrada n é zero.

O código seguinte apresenta um exemplo da utilização da metafunção factorial:

```
bool result = factorial<3>::value;
```

No código apresentado, é utilizada a metafunção factorial para calcular o factorial de 3. Como o resultado do factorial de 3 é calculado em tempo de compilação, a variável result é inicializada a 6.

A figura 5.1 apresenta um diagrama com as iterações necessárias para calcular o factorial de 3 em tempo de compilação:

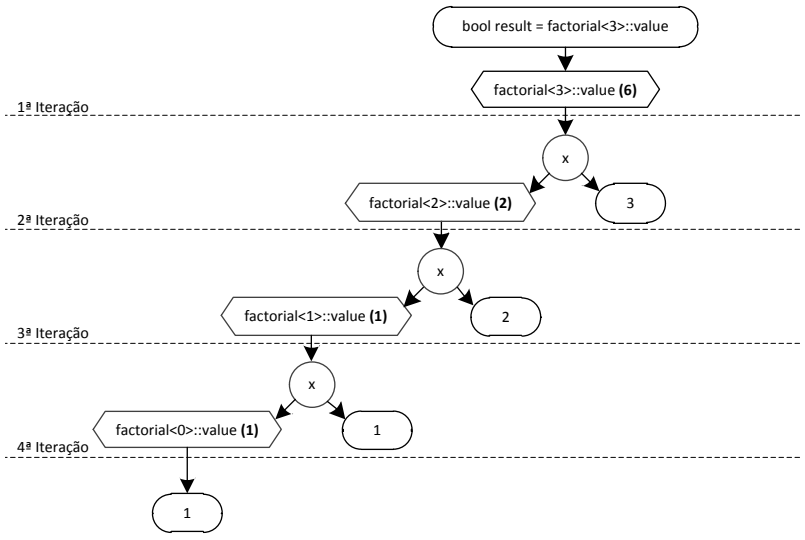


Figura 5.1: Diagrama com as iterações necessárias para calcular o factorial de 3.

Tal como a diagrama da figura 5.1 ilustra, são necessárias 4 iterações para resolver o valor do factorial de 3. Para calcular o factorial de 3, o factorial de 2 e o factorial de 1 é utilizada a *template* genérica factorial. Na última iteração, para calcular o factorial de 0 é utilizada a *template* especializada factorial (a condição de paragem da função recursiva factorial).

5.3 *Template Metaprogramming Frameworks*

Nesta secção, são apresentadas as *frameworks* mais conhecidas no mundo académico e na indústria, que utilizam *template metaprogramming* para gerir a variabilidade do código de um sistema.

A tabela 5.1 apresenta as *frameworks* que utilizam *template metaprogramming* e uma breve descrição de cada uma delas:

De seguida, são explicadas com mais detalhe as *frameworks* Boost.MPL (*MetaProgramming Library*) e GIL (*Generic Image Library*). A razão para explicar estas duas *frameworks* em detrimento das outras, prende-se com o facto de a primeira servir de base de implementação da maioria das *frameworks* TMP e a segunda implementar algoritmos genéricos de processamento de imagem relacionados com a área de videovigilância.

Tabela 5.1: *Frameworks* que utilizam *template metaprogramming*.

Nome	Descrição
Boost.MPL	A biblioteca Boost.MPL utiliza *template metaprogramming* para implementar em tempo de compilação sequências, algoritmos e metafunções.
Boost.Xpressive	A biblioteca Boost.Xpressive utiliza *template metaprogramming* para implementar expressões regulares.
Boost.Spirit	A biblioteca Boost.Spirit utiliza *template metaprogramming* para implementar gramáticas e descrições de formatos semelhantes ao formato *Extended Backus Naur Form* (EBNF), permitindo criar *parser-generators*.
Boost.Proto	A biblioteca Boost.Proto utiliza *template metaprogramming* para implementar *Embedded Domain-Specific Languages* in C++.
Blitz++	A biblioteca Blitz++ utiliza *template metaprogramming* para implementar objetos relacionados com computação científica de alto desempenho, tais como, *dense arrays*, *dense* vetores e geradores de números aleatórios.
Metaparse	A biblioteca *Metaparse* utiliza *template metaprogramming* para suportar a criação de *parsers* que fazem o *parsing* em tempo de compilação.
GIL	A biblioteca *Generic Image Library* utiliza *template metaprogramming* para abstrair a representação das imagens dos algoritmos de processamento de imagem.
MTL	A biblioteca *Matrix Template Library* utiliza *template metaprogramming* para implementar componentes de alto desempenho que permitem aplicar funcionalidades de álgebra linear numa grande variedade de formatos de matrizes.

5.3.1 *Boost Metaprogramming Library*

A biblioteca Boost.MPL (*MetaProgramming Library*) é um exemplo de uma biblioteca de uso genérico que utiliza C++ *template metaprogramming* para implementar em tempo de compilação sequências, algoritmos e metafunções (Gurtovoy and Abrahams).

Tal como existe a biblioteca STL (*Standard Template Library*) que fornece um conjunto de classes de uso genérico, que podem ser utilizadas por qualquer tipo de dados para realizar operações em tempo de execução, a biblioteca Boost.MPL fornece um conjunto de sequências e algoritmos de uso genérico que podem ser

executados por metafunções em tempo de compilação.

A biblioteca Boost.MPL desempenha um papel preponderante no desenvolvimento de novas *frameworks* que utilizam *template metaprogramming*. Normalmente, a biblioteca Boost.MPL serve de base de implementação de novas sequências, algoritmos e metafunções definidas nas novas *frameworks*. Exemplos disso, são a *framework* da *Adobe* GIL (*Generic Image Library*) e a *framework Metaparse*.

De seguida, são apresentadas algumas das funcionalidades fornecidas pela biblioteca boost.MPL, tais como, sequências, iteradores, algoritmos, metafunções, tipos de dados e macros.

Sequências

A biblioteca Boost.MPL define vários tipos de sequências, tais como vetores, listas e conjuntos, que permitem agrupar tipos de dados que só existem em tempo de compilação. Como muitos dos algoritmos da biblioteca utilizam sequências como fonte de dados, as sequências desempenham um papel de extrema importância na biblioteca.

Para exemplificar como é definida uma sequência do tipo vetor, onde cada elemento do vetor é um tipo de dados, é apresentado o seguinte código:

```
// define um vetor com os tipos de dados
typedef mpl::vector<char, short, int, long, float, double> types;
```

Na definição do vetor types é utilizada a sequência vector da biblioteca Boost.MPL, que é constituído por seis elementos: char, short, int, long, float e double.

Iteradores

Os iteradores permitem endereçar de forma genérica um determinado elemento de uma sequência. Essencialmente, todos os algoritmos da biblioteca que operam com sequências utilizam iteradores.

Para exemplificar a utilização de iteradores é apresentado o seguinte código:

```
// define um vetor com os tipos de dados do tipo unsigned
typedef vector< unsigned char,unsigned short, unsigned int,unsigned long > unsigned_types;

// o tipo de dados iter contém o endereço do primeiro iterador da sequência
typedef begin<unsigned_types>::type iter;
```

```
// verifica se o tipo de dados da referência do iterador é do tipo unsigned char
BOOST_MPL_ASSERT(( is_same< deref<iter>::type, unsigned char > ));
```

Primeiro é criado um vetor unsigned_types com todos os tipos de dados primitivos do C++ do tipo unsigned. De seguida, é utilizada a metafunção begin que retorna o primeiro iterador do vetor unsigned_types. Por fim, é verificado se o tipo de dados da referência do iterador iter é do tipo unsigned char.

Algoritmos

A biblioteca Boost.MPL disponibiliza um vasto número de algoritmos que satisfazem a maioria das necessidades relacionadas com o processamento de dados em sequências em tempo de compilação. Essencialmente, os algoritmos implementados na biblioteca estão muito relacionados com algoritmos implementados na biblioteca STL.

Algoritmos de Iteração Os algoritmos de iteração permitem executar operações pré-definidas, de forma iterativa, em cada um dos elementos de uma sequência. Assim, é possível abstrair o programador da complexidade associada aos detalhes de implementação relacionados com o processo de iteração de uma sequência. Um dos algoritmos de iteração mais utilizado da biblioteca Boost.MPL é o algoritmo fold.

O código seguinte apresenta a *sinopse* do algoritmo fold:

```
template<typename Sequence,
         typename State,
         typename ForwardOp>
struct fold {
      typedef unspecified type;
};
```

O algoritmo fold retorna o resultado da execução sucessiva da operação ForwardOp em cada um dos elementos da sequência Sequence, onde o estado inicial do processo iterativo é definido pelo tipo de dados State.

O código seguinte conta o número de elementos do tipo float no vetor types utilizando o algoritmo fold:

```
// define o vetor types com os tipos de dados a processar pelo algoritmo fold
typedef vector<long, float, short, double, float, long, long double> types;

// calcula o número de elementos do tipo float na sequência types
```

```
typedef fold<types,
             int_<0>,
             if_< is_float<_2>, next<_1>, _1 >
>::type number_of_floats;

// verifica se são 4 o número de elementos do tipo float na sequência types
BOOST_MPL_ASSERT_RELATION( number_of_floats::value, ==, 4 );
```

Numa primeira fase, é criado o vetor types que define uma sequência com sete tipos de dados. De seguida, o algoritmo de iteração fold recebe como parâmetros de entrada: o vetor types, a variável int_<0> e o construtor condicional if_<...>. O vetor types define a sequência com os tipos de dados a processar pelo o algoritmo de iteração fold. A variável int_<0> define o estado inicial do processo iterativo. O construtor condicional if_<...> utiliza a metafunção is_float para verificar se um determinado tipo de dados da sequência é do tipo float. Se o tipo de dados for do tipo float, a variável int_<0> é incrementada através da metafunção next, caso contrário o valor da variável não é alterado. Por fim, é utilizada a macro BOOST_MPL_ASSERT_RELATION para verificar se o valor da constante value do tipo de dados number_of_floats é 4.

Algoritmos de Consulta Os algoritmos de consulta permitem fazer consulta de dados em sequências. A maioria dos algoritmos de consulta utilizam o algoritmo de iteração fold apresentado anteriormente. Um dos algoritmos de consulta mais utilizado da biblioteca Boost.MPL é o algoritmo find.

O código seguinte apresenta a *sinopse* do algoritmo find:

```
template<typename Sequence,
         typename T>
struct find {
      typedef unspecified type;
};
```

O algoritmo find retorna o iterador da primeira ocorrência do tipo de dados T a pesquisar na sequência Sequence. Caso não seja encontrado o tipo de dados T na sequência Sequence, é devolvido um iterador que aponta para o fim da sequência.

O código seguinte pesquisa o elemento do tipo unsigned no vetor types utilizando o algoritmo find:

```
// define o vetor types com os tipos de dados a processar pelo algoritmo find
typedef vector<char, int, unsigned, long, unsigned long> types;

// procura o tipo de dados unsigned na sequência types
typedef find<types, unsigned>::type iter;
```

```
// verifica se o valor da posição do iterador é 2
BOOST_MPL_ASSERT_RELATION( iter::pos::value, ==, 2 );
```

Numa primeira fase, é criado o vetor types que define uma sequência com cinco tipos de dados. De seguida, o algoritmo de consulta find recebe como parâmetros de entrada: o vetor types e o tipo de dados unsigned. O vetor types define a sequência com os tipos de dados a processar pelo o algoritmo de consulta find. O tipo de dados unsigned define o tipo de dados a pesquisar na sequência types. Por fim, é utilizada a macro BOOST_MPL_ASSERT_RELATION para verificar se o valor da posição do iterador devolvido pelo algoritmo de consulta find é 2.

Algoritmos de Transformação Os algoritmos de transformação permitem criar novas sequências ao executar operações de transformação em sequências já existentes. Um algoritmo de transformação típico recebe como parâmetro de entrada uma ou mais sequências e uma metafunção de transformação, e retorna uma nova sequência construída de acordo com a execução da metafunção de transformação na(s) sequência(s) de entrada. Um dos algoritmos de transformação mais utilizado da biblioteca Boost.MPL é o algoritmo copy.

O código seguinte apresenta a *sinopse* do algoritmo copy, que retorna uma cópia da sequência original Sequence.

```
template<typename Sequence,
         typename In = unspecified>
struct copy {
      typedef unspecified type;
};
```

O código seguinte cria uma cópia do vetor types utilizando o algoritmo copy:

```
// define o vetor de inteiros com os valores de 0 a 9
typedef vector_c<int, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9> numbers;

// copia os elementos do vetor numbers para o vetor result
typedef copy<numbers>::type result;

// verifica se o número de elementos do vetor result é 10
BOOST_MPL_ASSERT_RELATION( size<result>::value, ==, 10 );
```

Numa primeira fase, é criado o vetor numbers que define uma sequência de dez elementos com os valores de 0 a 9. De seguida, o algoritmo de transformação copy recebe o vetor numbers como parâmetro de entrada e coloca o resultado no tipo de dados result. Por fim, é utilizada a macro BOOST_MPL_ASSERT_RELATION para verificar se o tamanho da nova sequência result é 10.

Algoritmos de *Runtime* Os algoritmos de *runtime* permitem executar determinadas tarefas em sequências em *runtime*. O único algoritmo de *runtime* definido na biblioteca boost.MPL é o algoritmo for_each.

O código seguinte apresenta a *sinopse* do algoritmo de *runtime* for_each:

```
template<typename Sequence,
         typename F>
void for_each( F f );

template<typename Sequence,
         typename TransformOp,
         typename F>
void for_each( F f );
```

A biblioteca boost.MPL define duas assinaturas para o algoritmo de *runtime* for_each. Na primeira, a função *runtime* f é executada em todos os elementos da sequência Sequence. Na segunda, a função *runtime* f é executada no resultado da execução do algoritmo de transformação TransformOp em cada um dos elementos da sequência Sequence.

O código seguinte imprime no ecrã em tempo de execução, todos os elementos de um vetor utilizando o algoritmo for_each:

```
// define a metafunção value_printer
struct value_printer {
      template< typename U > void operator()(U x) {
            std::cout << x << 'n';
      }
};

int main()
{
      // chama a metafunção value_printer para cada elemento do vetor
      for_each< range_c<int,0,10> >( value_printer() );
}
```

O exemplo apresentado define a metafunção value_printer que através do operador () permite imprimir no ecrã o valor da variável x em tempo de execução. Por fim, é chamado o algoritmo de *runtime* for_each que recebe como parâmetro de entrada de *template* um vetor com os números de 0 a 10, e como parâmetro de entrada da função o operador () da estrutura value_printer.

Metafunções

Uma metafunção é uma classe ou classe *template* que representa uma função que é executada em tempo de compilação. Todos os argumentos de uma metafunção

são tipos de dados e são passados através dos parâmetros de entrada da *template*. O resultado da execução da metafunção é acessível através da instanciação de tipos de dados definidos dentro da *template* (*nested types*) utilizando typedef ou constantes. Existem dois tipos de metafunções: as metafunção *non-nullary* e as metafunções *nullary*. As metafunções *non-nullary* recebem parâmetros de entrada de *template* (argumentos de entrada da metafunção). As metafunções *nullary* não recebem parâmetros de entrada de *template* e definem membros dentro da classe *template* do tipo typedef e/ou constantes.

A biblioteca boost.MPL define um conjunto de metafunção que podem ser classificadas em duas categorias: metafunção de uso genérico e metafunções numéricas. As metafunções de uso genérico são utilizadas para implementar seleção de tipos, invocar metafunção *higher-order* e fazer *binding* de argumentos. As metafunções numéricas são utilizadas em operações aritméticas, comparações, operações lógicas.

Metafunções de Uso Genérico A metafunção if_ define um exemplo de uma metafunção de uso genérico que permite selecionar tipos. O código seguinte apresenta a *sinopse* da metafunção if_:

```
template<typename C,
         typename T1,
         typename T2>
struct if_ {
      typedef unspecified type;
};
```

A metafunção if_ recebe três parâmetros de entrada: o tipo de dados C que define a condição, o tipo de dados T1 que define o tipo de dados que é retornado pela metafunção se a condição C for verdade e o tipo de dados T2 que define o tipo de dados que é retornado pela metafunção se a condição C for falsa. Basicamente, a metafunção if_ implementa o construtor condicional if em tempo de compilação.

O código seguinte apresenta um exemplo da utilização da metafunção if_:

```
// define o tipo de dados t1 com o resultado da metafunção if_
typedef if_<true_, char, long>::type t1;

// verifica se t1 é do tipo de dados char
BOOST_MPL_ASSERT(( is_same<t1, char> ));
```

No exemplo apresentado, é definido o tipo de dados t1 com o resultado da metafunção if_. A metafunção if_ recebe como argumentos da metafunção: o tipo de dados

true_ como condição, o tipo de dados char como o tipo de dados da condição then e o tipo de dados long como o tipo de dados da condição else. Por fim, é verificado se o tipo de dados t1 é do tipo char.

Metafunções Numéricas A metafunção plus define um exemplo de uma metafunção numérica que devolve o resultado da soma de todos os argumentos. O código seguinte apresenta a *sinopse* da metafunção plus:

```
template<typename T1,
         typename T2,
         typename T3 = unspecified,
         ...,
         typename Tn = unspecified>
struct plus {
      typedef unspecified type;
};
```

Como a operação soma necessita no mínimo de dois operadores, a metafunção plus recebe no mínimo dois parâmetros de entrada de *template*: o parâmetro de entrada de *template* T1 e o parâmetro de entrada de *template* T2.

O código seguinte apresenta um exemplo da utilização da metafunção plus:

```
// define o tipo de dados r com o resultado da metafunção plus
typedef plus< int_<−10>, int_<3>, long_<1> >::type r;

// verifica se o valor da constante value de r é −6
BOOST_MPL_ASSERT_RELATION( r::value, ==, −6 );
```

No exemplo apresentado, é definido o tipo de dados r com o resultado da metafunção plus. A metafunção plus recebe como argumentos da metafunção: o tipo de dados int_<-10>, o tipo de dados int_<3> e o tipo de dados int_<1>. Por fim, é verificado se o valor da soma dos três argumentos de entrada da *template* é -6.

Tipo de Dados

A biblioteca boost.MPL classifica os tipos de dados em duas categorias: os tipos de dados numéricos e os tipos de dados universais.

Tipos de Dados Numéricos Os tipos de dados numéricos definem estruturas de dados que representam variáveis em que a base é um número. Um dos tipos de dados numéricos mais utilizados é o tipo de dados int_. O código seguinte apresenta a *sinopse* do tipo de dados int_:

```
template<int N>
struct int_ {
      // unspecified
      // ...
};
```

O código seguinte apresenta um exemplo da utilização do tipo de dados int_:

```
// define o tipo de dados r com o resultado da metafunção plus
typedef plus< int_<−10>, int_<3>, long_<1> >::type r;

// verifica se o valor da constante value de r é −6
BOOST_MPL_ASSERT_RELATION( r::value, ==, −6 );
```

No exemplo apresentado, é definido o tipo de dados r com o resultado da metafunção plus. A metafunção plus recebe como argumentos da metafunção: o tipo de dados int_<-10>, o tipo de dados int_<3> e o tipo de dados int_<1>. Por fim, é verificado se o valor da soma dos três argumentos de entrada da *template* é -6.

Tipos de Dados Universais Os tipos de dados universais definem estruturas de dados que representam dados genéricos. Um dos tipos de dados universais mais utilizados é o tipo de dados pair. O código seguinte apresenta a *sinopse* do tipo de dados pair:

```
template<typename T1,
         typename T2>
struct pair {
      typedef pair type;
      typedef T1 first;
      typedef T2 second;
};
```

o tipo de dados pair cria um par de dados, onde o tipo de dados T1 representa o primeiro tipo de dados do par e o tipo de dados T2 representa o segundo tipo de dados do par.

O código seguinte apresenta um exemplo da utilização do tipo de dados pair:

```
// define o tipo de dados p como do tipo pair
typedef pair< int_<1>, int_<2> >::type p;

// verifica se o valor do tipo de dados first do tipo de dados p é 1
BOOST_MPL_ASSERT_RELATION( p::first::value, ==, 1 );
// verifica se o valor do tipo de dados second do tipo de dados p é 2
BOOST_MPL_ASSERT_RELATION( p::second::value, ==, 2 );
```

No exemplo apresentado, o tipo de dados p é do tipo pair com os argumentos de entrada 1 e 2. De seguida, é verificado se o valor do campo first do tipo de dados p

é 1. Por fim, é verificado se o valor do campo second do tipo de dados p é 2.

Macros

Embora a utilização de macros não seja do agrado de muitos programadores, o que é certo é que as macros desempenham um papel preponderante na metaprogramação em C++. Isso, deve-se ao facto de algumas das funcionalidades não serem possíveis implementar com *templates* ou *template metaprogramming* em tempo de compilação.

A biblioteca boost.MPL fornece um conjunto de macros que permitem fazer verificações estáticas, introspeção de dados e configurar dados. As macros mais utilizadas na biblioteca são as macros que permitem fazer verificações estáticas, tal como a macro BOOST_MPL_ASSERT. O código seguinte apresenta a *sinopse* da macro BOOST_MPL_ASSERT:

```
#define BOOST_MPL_ASSERT( pred ) \
      unspecified token sequence \
/**/
```

A macro BOOST_MPL_ASSERT verifica se a condição pred é verdadeira. Caso seja falsa, é gerado um erro de compilação a explicar o sucedido.

O código seguinte apresenta um exemplo da utilização da macro BOOST_MPL_ASSERT:

```
// verifica se os tipos de dados T e U são do mesmo tipo
BOOST_MPL_ASSERT(( is_same< T,U > ));
```

No exemplo apresentado, a macro BOOST_MPL_ASSERT verifica se os tipos de dados T e U são do mesmo tipo, utilizando a metafunção is_same.

5.3.2 *Generic Image Library*

O elevado grau de variabilidade existente nos formatos que permitem representar uma imagem, faz com que seja muito difícil implementar algoritmos de processamento de imagem genéricos, sem pôr em causa a eficiência dos mesmos. Para tentar resolver este problema, a *Adobe* desenvolveu a biblioteca GIL (*Generic Image Library*), com o intuito de criar uma biblioteca genérica que faça a abstração da representação das imagens dos algoritmos de processamento de imagem sem afectar o desempenho dos mesmos.

A figura 5.2 apresenta a variabilidade que existe nos formatos de representação de uma imagem:

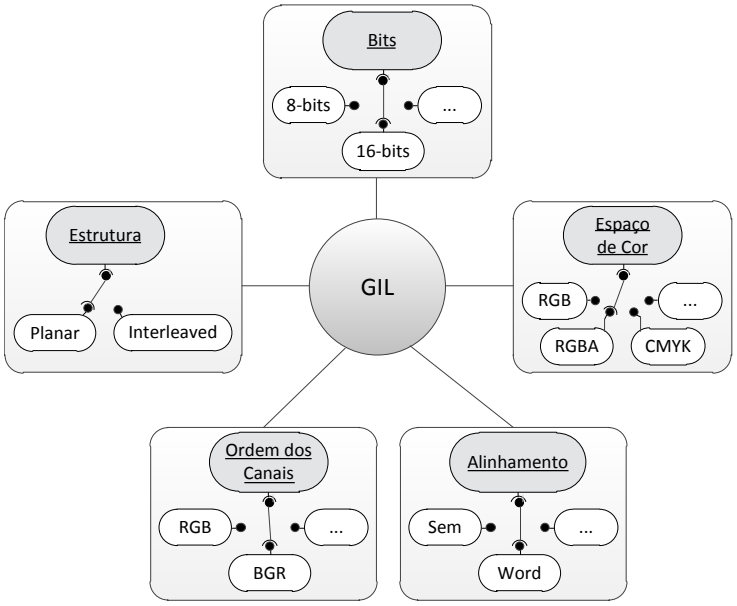


Figura 5.2: Variabilidade que existe nos formatos de representação de uma imagem.

Tal como ilustra o diagrama da figura 5.2, a biblioteca GIL fornece modelos para imagens que variam: (i) na estrutura (planar vs. interleaved), (ii) no espaço de cor e a presença do canal alpha (RGB, RGBA, CMYK, etc), (iii) no número de *bits* de cada componente da imagem (8-bit, 16-bit, etc), na ordem dos canais (RGB vs. BGR, etc) e na estratégia de alinhamento das linhas da imagem (sem alinhamento, alinhado à *word*, etc).

Espaço de Cor

O espaço de cor representa um modelo abstrato de descrever a forma como as cores são representadas em tuplos de números. Normalmente, os tuplos são constituídos por três ou quatro componentes de cor. A biblioteca GIL fornece suporte para os seguintes espaços de cor: `gray_t`, `rgb_t`, `rgba_t`, and `cmyk_t`.

O código seguinte apresenta como é definido o espaço de cor `rgb_t`:

```
// estrutura de dados que representa a componente de cor vermelha
struct red_t{};
// estrutura de dados que representa a componente de cor verde
struct green_t{};
// estrutura de dados que representa a componente de cor azul
struct blue_t{};
```

```
// vetor com as componentes de cor que representam o espaço de cor RGB
typedef mpl::vector3<red_t,green_t,blue_t> rgb_t;
```

Para definir o espaço de cor RGB através do tipo de dados rgb_t, é necessário definir em primeiro lugar o tipo de dados para cada uma das componentes de cor utilizadas no espaço de cor. Assim, são definidas as estruturas de dados red_t para a componente de cor vermelha, green_t para a componente de cor verde e blue_t para a componente de cor azul. Depois de definidas as componentes de cor, é definido o vetor de três elementos rgb_t com as componentes de cor red_t, green_t e blue_t.

A biblioteca GIL também fornece suporte para espaços de cor sem nome com n componentes, que vão desde duas até cinco componentes: devicen_t<2>, devicen_t<3>, devicen_t<4> e devicen_t<5>.

Ordem dos Canais

A ordem dos componentes de cor na definição do espaço de cor especifica a ordem semântica de cada componente no espaço de cor. Por exemplo, a componente de cor vermelha é a primeira componente semântica no espaço de cor RGB, mas pode ocupar uma outra posição na memória.

O código seguinte define a estrutura de dados layout que permite definir a ordem das componentes de cor de cada espaço de cor:

```
template <typename ColorSpace,
          typename ChannelMapping = mpl::range_c<int,0,mpl::size<ColorSpace>::value> >
struct layout {
    typedef ColorSpace color_space_t;
    typedef ChannelMapping channel_mapping_t;
};
```

A estrutura de dados layout recebe dois parâmetros de entrada de *template*: o tipo de dados ColorSpace e o tipo de dados ChannelMapping. O tipo de dados ColorSpace define o espaço de cor através do tipo de dados da estrutura color_space_t. O tipo de dados ChannelMapping define o ordem de cada componente no vetor que representa o espaço de cor através do tipo de dados da estrutura channel_mapping_t. Por defeito o tipo de dados ChannelMapping é uma sequência do tipo mpl::range_c que define um vetor com tantos elementos quantos o número de componentes de cor do espaço de cor, preenchido de com 0 no primeiro elemento do *array*, 1 no segundo, e assim sucessivamente.

Para definir as diferentes *layouts* (ordem dos componentes de cor) do espaço de cor RGB, são necessárias as seguintes linhas de código:

```
typedef layout<rgb_t> rgb_layout_t;
typedef layout<rgb_t, mpl::vector3_c<int,2,1,0> > bgr_layout_t;
typedef layout<rgb_t, mpl::vector3_c<int,1,2,0> > gbr_layout_t;
```

O código apresentado define três *layouts* do espaço de cor RGB: o *layout* rgb_layout_t, o *layout* bgr_layout_t e o *layout* gbr_layout_t. Para definir o *layout* rgb_layout_t é utilizada a estrutura de dados layout com o parâmetro de entrada do espaço de cor rgb_t. A ordem de cada uma das componentes não é especificada porque é a ordem por defeito (0, 1, 2 -> RGB). Para definir o *layout* bgr_layout_t é utilizada a estrutura de dados layout com o parâmetro de entrada do espaço de cor rgb_t e a ordem dos componentes de cor (2, 1, 0). Por fim, para definir o *layout* gbr_layout_t é utilizada a estrutura de dados layout com o parâmetro de entrada do espaço de cor rgb_t e a ordem dos componentes de cor (1, 2, 0).

Número de *Bits* por Componente de Cor

Cada componente de cor pode ser representada por um número variável de *bits*. O formato mais comum, é cada componente de cor ser representada por 8 *bits*, isto é, o valor de cada componente de cor pode variar de 0 a 255.

O código seguinte apresenta os vários tipos de dados que representam o número de *bits* de cada componente de cor definida na biblioteca GIL:

```
// componente de cor com 8 bits sem sinal
typedef boost::uint8_t bits8;
// componente de cor com 16 bits sem sinal
typedef boost::uint16_t bits16;
// componente de cor com 32 bits sem sinal
typedef boost::uint32_t bits32;
// componente de cor com 8 bits com sinal
typedef boost::int8_t bits8s;
// componente de cor com 16 bits com sinal
typedef boost::int16_t bits16s;
// componente de cor com 32 bits com sinal
typedef boost::int32_t bits32s;
```

O código apresentado define seis tipos de dados que representam o número de *bits* de cada componente de cor, três deles do tipo *unsigned* (sem sinal) e os outros três do tipo *signed* (com sinal). Essencialmente, cada um deles permitem especificar qual o valor mínimo e máximo que cada componente de cor pode ter. Por exemplo, se for utilizado o tipo de dados bits8, cada componente de cor pode ir de 0 até 255.

Estrutura da Imagem em Memória

Existem dois tipos comuns de estruturar uma imagem em memória: a estrutura planar e a estrutura *interleaved*. A estrutura planar organiza em memória os dados de cada componente de cor em planos diferentes. Contrariamente à anterior, a estrutura *interleaved* organiza em memória os dados de cada componente de cor em grupos.

Para estruturar uma imagem em memória a biblioteca GIL define duas vistas, tal como apresenta o seguinte código:

```
// define a vista interleaved
template <typename Iterator>
typename type_from_x_iterator<Iterator>::view_t
interleaved_view(point2<std::size_t> dim,
                 Iterator pixels, std::ptrdiff_t rowsize_in_bytes) {
    ...
}

// define a vista planar
template <typename Iterator>
typename type_from_x_iterator<Iterator>::view_t
planar_view(point2<std::size_t> dim,
                 Iterator pixels, std::ptrdiff_t rowsize_in_bytes) {
    ...
}
```

Assim, quando é instanciada a vista interleaved_view é criada em memória uma imagem com a estrutura *interleaved*, quando é instanciada a vista planar_view é criada em memória uma imagem com a estrutura planar.

Alinhamento de Memória

Ao alocar espaço em memória para uma imagem, é necessário ter em consideração um conjunto de fatores, sendo um deles o alinhamento de memória. O alinhamento de memória está relacionado com o facto de o endereço de memória da imagem ser múltiplo do tamanho do tipo de dados word da máquina. No caso específico de ser uma máquina de 32-*bits*, o tamanho de uma word é de 4 *bytes*, logo para a memória estar alinhada o endereço de memória tem que ser múltiplo de 4. Assim, sempre que a memória de uma imagem estiver alinhada, faz com que aumente o desempenho nos acessos de leitura e escrita da memória.

A biblioteca GIL define a classe image que representa uma imagem em memória:

```
template <typename Pixel, bool IsPlanar, typename Alloc=std::allocator<unsigned char> >
```

```
class image {
        ...
        image(const point_t& dimensions, std::size_t alignment=1)
                        : _memory(0), _align(alignment) {
                ...
        }
        ...
};
```

No construtor da classe image é passado como parâmetro de entrada o parâmetro alignment que especifica qual o alinhamento de memória utilizado. Por defeito é o valor 1, indicando que a imagem é criada sem alinhamento, o que implica que quando se muda de linha não são adicionados *bits* de padding para alinhar a próxima linha de *bits* que compõem a imagem.

Instanciação de uma Imagem

Tal como foi referido anteriormente, a biblioteca GIL permite criar imagens que são representadas em vários tipos de formatos sem afetar o desempenho dos algoritmos de processamento de imagem. Para exemplificar como se cria uma imagem utilizando a biblioteca GIL e se aplicam algoritmos de processamento de imagem à imagem, é apresentado o seguinte código:

```
// lê uma imagem do ficheiro "monkey.jpg"
jpeg_read_image("monkey.jpg", img);
// cria uma vista da imagem
step1 = view(img);
// cria uma sub-imagem da imagem anterior
step2 = subimage_view(step1, 200,300, 150,150);
// converte a imagem do formato rgb para o formato gray
step3 = color_converted_view<rgb8_view_t,gray8_pixel_t>(step2);
// roda a imagem 180 graus
step4 = rotated180_view(step3);
// faz o sub sampling da imagem no eixo dos xx para metade
step5 = subsampled_view(step4, 2,1);
// escreve a imagem final no ficheiro "monkey_transform.jpg"
jpeg_write_view("monkey_transform.jpg", step5);
```

A figura 5.3 apresenta as imagens de cada um dos estágios intermédios de processamento:

O exemplo apresentado foi extraido do site da biblioteca GIL (Bourdev). No estágio 1, é lida a imagem do ficheiro ”monkey.jpg” através da função jpeg_read_image, que recebe como parâmetro de entrada o nome do ficheiro a ler e o objeto que representa a imagem em TMP. No estágio 2, é criada uma imagem reduzida da

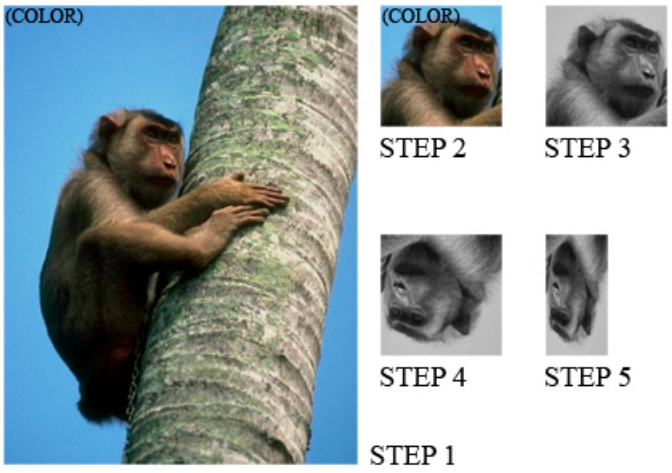


Figura 5.3: Imagens de cada um dos estágios intermédios de processamento. Imagem extraída de (Bourdev).

imagem inicial. No estágio 3, é feita a conversão do formato de cor RGB para o formato de cor em tons de cinzento. No estágio 4, é feita a rotação da imagem com um grau de 180 graus. No estágio 5, é feito o *sub-sampling* da componente do eixo dos xx para metade. E por fim, é gravada a imagem final no ficheiro "monkey_transform.jpg".

Ao compilar o código apresentado, o compilador gera o código totalmente otimizado para o fim proposto tal como se este fosse escrito totalmente à mão. Assim, é gerado menos código fazendo com que o desempenho aumente e o tamanho do ficheiro executável diminua.

5.4 Conclusão

Neste capítulo foi apresentado o *template metaprogramming* (TMP) como a principal técnica da linguagem de programação C++ para resolver a variabilidade do código de um sistema em tempo de compilação. Depois de feita uma breve introdução do TMP e dos blocos básicos de código que compõem os dados e os algoritmos TMP, foram apresentadas duas *frameworks*: a *framework* Boost.MPL e a *framework* GIL (*Generic Image Library*) da *Adobe*. Em cada uma delas foi descrita a forma como é gerida a variabilidade existente ao nível do código.

Depois de analisadas as duas *frameworks*, chegou-se à conclusão que embora o código TMP (código da linguagem funcional) seja muito mais difícil de implemen-

tar e compreender comparativamente com o código C++ tradicional (código da linguagem imperativa), o código executável final da implementação em TMP é muito mais otimizado que o código da implementação em C++. Assim, conclui-se que o TMP é a técnica ideal para resolver a variabilidade do código de sistemas complexos em tempo de compilação, não incorrendo em custos de *overhead* no que diz respeito ao desempenho e ao tamanho do ficheiro executável de uma aplicação, especialmente quando estes são compilados para serem executados em sistema embebidos.

Capítulo 6

Framework Generativa de Videovigilância

6.1 Introdução

No capítulo anterior, foi apresentado o *template metaprogramming* (TMP), como a principal técnica da linguagem de programação C++, para resolver a variabilidade do código de um sistema, em tempo de compilação. Depois de apresentados os blocos básicos da programação TMP e as principais vantagens do uso desta técnica na gestão da variabilidade do código, foram apresentadas as principais *frameworks* desenvolvidas pela comunidade científica e pela indústria, que utilizam TMP para a gestão da variabilidade do código.

Neste capítulo, são apresentadas as fases de análise, desenho e implementação de uma *framework* generativa para sistemas de videovigilância. Nas fases de análise e desenho, são utilizas técnicas de *Software Product Line* (SPL) e *Agile*, para gerir a variabilidade das funcionalidades do sistema ao nível da modelação. Por fim, para gerir a variabilidade das funcionalidades do sistema ao nível da implementação, foram desenvolvidos componentes totalmente reconfiguráveis, utilizando C++ *template metaprogramming*.

Como o código TMP é bastante complexo, é apresentada a ferramenta de modelação TMPxML. A ferramenta de modelação TMPxML, permite gerar de forma automática, código TMP a partir de modelos TMPxML. Ao utilizar modelos TMPxML, o programador cria código TMP de uma forma mais rápida e eficaz,

comparativamente com o método manual de desenvolvimento de código TMP.

6.2 *Software Product Line*

No processo de análise, desenho e implementação da *framework* de videovigilância apresentada neste capítulo, foram utilizadas técnicas de desenvolvimento de *software* de linha de produto. Pretendeu-se, com a utilização destas técnicas, que o foco de desenvolvimento não esteja direcionado apenas para um produto específico mas sim para uma família de produtos da videovigilância. Assim, numa primeira fase, domínio de engenharia, é necessário identificar e implementar os componentes que são comuns e os componentes que são variáveis na linha de produção. Este processo permite desenvolver componentes que são reutilizáveis na produção de um produto específico da linha de produção ("*developing for reuse*"). Numa segunda fase, engenharia da aplicação, são selecionados de um repositório de componentes, os componentes necessários para obter o ficheiro executável final para responder a uma configuração específica ("*developing with reuse*"). Ao explorar as funcionalidades comuns dos vários tipos de produtos que compõem a família de produtos é possível aumentar a produtividade e a qualidade dos produtos (Weiss and Lai, 1999). Na secção que se segue são descritas em detalhe estas duas fases no contexto do desenvolvimento da *framework* do sistema de videovigilância.

6.2.1 Domínio de Engenharia

Análise do Domínio

A fase de análise do domínio pode ser dividida em duas etapas: o âmbito do domínio e a modelação das funcionalidades.

Âmbito do Domínio O âmbito do domínio determina quais os sistemas e funcionalidades que fazem parte do domínio. Na Figura 6.1 é apresentado o domínio da videovigilância.

No âmbito do domínio da videovigilância são englobados cinco subsistemas: sistema de captura de áudio/vídeo, sistema de processamento de imagem, sistema

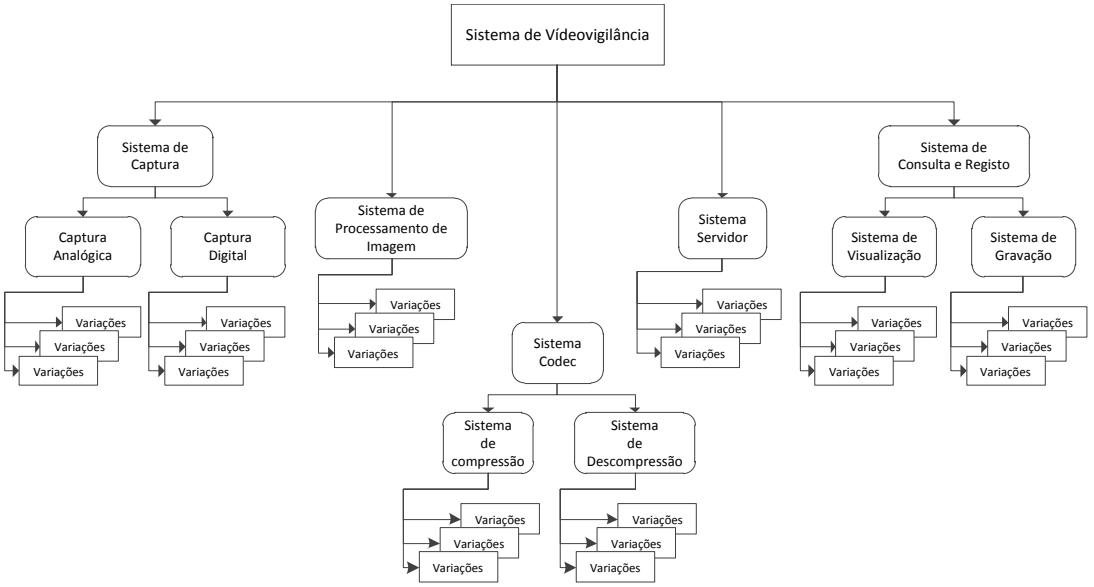


Figura 6.1: Âmbito do Domínio da Videovigilância.

de compressão/descompressão (*codec*), sistema servidor e sistema de consulta e registo.

O sistema de captura de áudio/vídeo, por sua vez, é constituído por dois subsistemas, o sistema de captura analógica e o sistema de captura digital. O sistema de captura analógica implementa todas as funcionalidades necessárias para fazer captura de dados (áudio/vídeo) de dispositivos analógicos em vários sistemas operativos, tais como o *DirectShow* no sistema operativo Windows e o V4L2 (*Video for Linux*) no sistema operativo Linux. O sistema de captura digital implementa as funcionalidades que permitem fazer a captura de áudio e vídeo de câmaras IP de vários fabricantes que suportam vários protocolos de transmissão.

O sistema de processamento de imagem implementa funcionalidades que processam as imagens provenientes dos sistemas de captura. As funcionalidades mais comuns em sistemas de videovigilância são: deteção de movimento, reconhecimento facial, seguimento de objetos e reconhecimento de matrículas.

O sistema de *codec* implementa funcionalidades relacionadas com a compressão e descompressão de dados. Os dados a comprimir ou a descomprimir são dados de áudio e de vídeo. Logo, o sistema implementa os *codecs* mais comuns nos sistemas de videovigilância, tais como o *codec* MJPEG, o *codec* MPEG-4 e o *codec* H.264.

O sistema servidor implementa funcionalidades que permitem, não só o controlo remoto do sistema de videovigilância, mas também a visualização remota em tempo

real das imagens capturadas pelas câmaras e a reprodução dos vídeo gravados.

Finalmente, o sistema de consulta e registo implementa funcionalidades que permitem consultar e registar eventos de dados (áudio/vídeo) do sistema de videovigilância. Essencialmente, o sistema de consulta e registo implementa as funcionalidades de visualização local e as funcionalidades de gravação.

Modelação das Funcionalidades A modelação das funcionalidades identifica as funcionalidades que são comuns e as que são variáveis no domínio da videovigilância bem como a dependência entre as funcionalidades variáveis. O modelo de funcionalidades é representado através de diagramas de funcionalidade. No caso da videovigilância é definido um *template* artefacto para o *filter graph* e um *template* artefacto para cada filtro. Na próxima secção são apresentados o diagrama de funcionalidade do *filter graph* e um exemplo de um diagrama de funcionalidades para cada um dos tipos de filtros.

Diagrama de Funcionalidades do *Filter Graph* O diagrama de funcionalidades do *filter graph* permite identificar e classificar os filtros que compõem o *filter graph* e as possíveis ligações entre eles. O diagrama de funcionalidades do *filter graph* está representado na Figura 6.2.

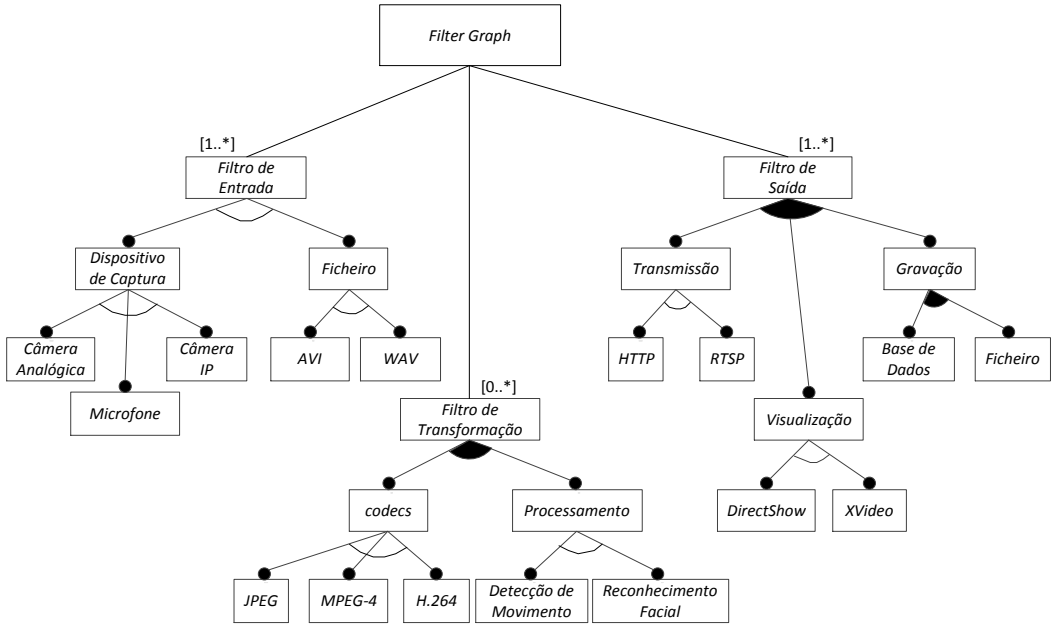


Figura 6.2: Diagrama de funcionalidades do *Filter Graph*.

O nó raiz representa o conceito (*Filter Graph*) que é composto por três funci-

onalidades: *Filtro de Entrada*, *Filtro de Transformação* e *Filtro de Saída*. As funcionalidades apresentadas são os tipos de filtros que compõem o *filter graph*.

As funcionalidades *Filtro de Entrada* e *Filtro de Saída* têm cardinalidade [1..*], o que significa que o *filter graph* pode ser composto por um ou mais filtros destes tipos. O *Filtro de Transformação* tem cardinalidade [0..*] que indica que o filtro é opcional (pode ter uma ou mais instâncias no *filter graph*). O número de instâncias do filtro de transformação depende da necessidade de fazer transformações intermédias nos dados de modo a permitir estabelecer a ligação entre o filtro de entrada e o filtro de saída.

A funcionalidade *Filtro de Entrada* é constituída por duas outras funcionalidades alternativas: *Dispositivo de Captura* e *Ficheiro*. Estas funcionalidades são alternativas, porque normalmente não é incumbida a um filtro de entrada a tarefa de realizar simultaneamente a captura de dados de dispositivos *live*, tais como câmaras e microfones, e a leitura de dados de ficheiros gravados em disco.

A funcionalidade *Filtro de Transformação* é constituída por duas funcionalidades cumulativas: *Codecs* e *Processamento*. As funcionalidades são cumulativas, porque para determinadas configurações do sistema por exemplo, deteção de movimento, é necessário ter um *codec* para fazer a compressão e descompressão de dados e posteriormente o processamento dos mesmos. Cada uma destas funcionalidades também apresenta variabilidade. Por exemplo, no caso dos *codecs* os mais importantes a implementar são o JPEG, MPEG-4 e H.264; já no caso do processamento de imagem, os mais utilizados são os de deteção de movimento e de reconhecimento facial.

Finalmente, a funcionalidade *Filtro de Saída* é constituída por três funcionalidades cumulativas: *Transmissão*, *Visualização* e *Gravação*. As funcionalidades são cumulativas, porque é possível num sistema estar a visualizar as imagens, fazer gravações para consulta posterior e enviar os dados para visualização remota. A funcionalidade *Transmissão* apresenta variabilidade ao nível do protocolo de rede utilizado para a transmissão de dados, a funcionalidade *Visualização* é alternativa, porque depende do sistema operativo onde o sistema está a correr: Windows ou Linux. Finalmente, a funcionalidade *Gravação* é constituída por funcionalidades cumulativas, visto que é possível gravar ficheiros em disco e registar em base de dados a sua informação.

Diagrama de Funcionalidades de um Filtro de Entrada Para representar o diagrama de funcionalidades de um filtro de entrada é dado como exemplo o filtro de entrada V4L2. O filtro de entrada V4L2 permite fazer a captura de câmaras analógicas no sistema operativo Linux. A Figura 6.3 apresenta o diagrama de funcionalidades do filtro de entrada V4L2.

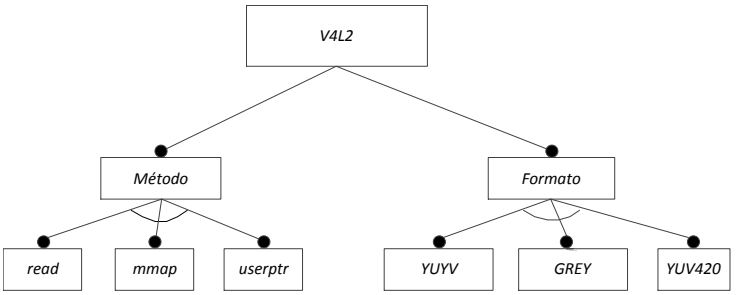


Figura 6.3: Diagrama de funcionalidades do filtro de entrada V4L2.

O nó raiz representa o conceito (*V4L2*), que é composto por duas funcionalidades obrigatórias: *Método* e *Formato*. A funcionalidade *Método* especifica o método usado para fazer a captura: *read*, *mmap* e *userptr*. O método *read* usa a chamada do sistema read, o método *mmap* usa memória mapeada para a captura de dados e o método *userptr* usa um *buffer* alocado pela própria aplicação. A funcionalidade *Formato* especifica o formato de captura. No diagrama de funcionalidades são apresentados apenas três formatos mas estão disponíveis mais formatos que dependem da placa de captura utilizada.

Diagrama de Funcionalidades de um Filtro de Transformação Para representar o diagrama de funcionalidades de um filtro de transformação, é dado como exemplo o filtro de transformação *JPEG Encoder*. O filtro de transformação *JPEG Encoder* permite fazer a compressão de imagens em formato descomprimido (*raw*) para o formato comprimido JPEG. A Figura 6.4 apresenta o diagrama de funcionalidades do filtro de transformação *JPEG Encoder*.

O nó raiz representa o conceito (*JPEG Encoder*), que é composto por cinco funcionalidades obrigatórias: a funcionalidade *Color Space* que representa o formato de imagem à entrada do filtro de transformação, a funcionalidade *Entropy* que representa o tipo de codificação utilizada para comprimir os dados da imagem, a funcionalidade *Compressão* que define qual o tipo de compressão JPEG utilizada, a funcionalidade *Sampling* que define o número de amostras de cada componente de cor que define a imagem e finalmente a funcionalidade *DCT* (*Discrete Cosine*

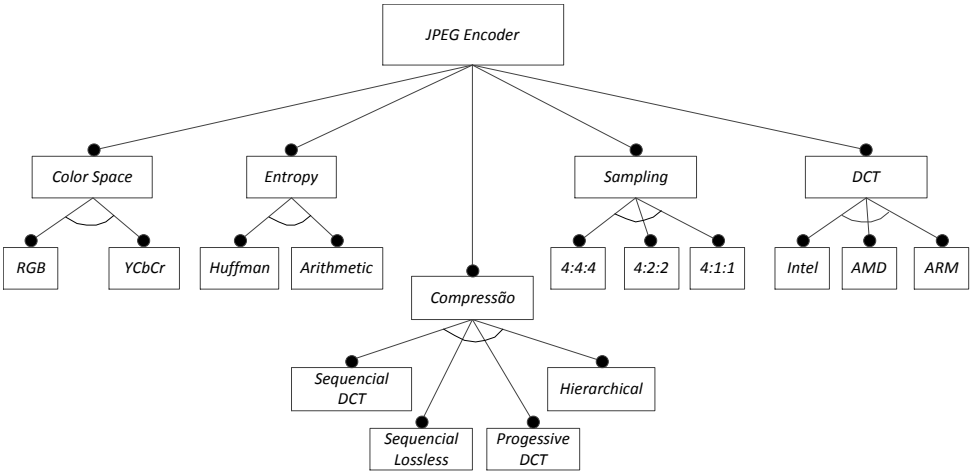


Figura 6.4: Diagrama de funcionalidades do filtro de transformação *JPEG Encoder*.

Transform) que representa a transformada discreta do co-seno que converte os píxeis da imagem numa matriz de coeficientes não correlacionados.

Os formatos mais comuns da funcionalidade *Color Space* são: o *RGB* e *YCbCr*. No caso do formato de entrada ser o RGB, é necessário converter o formato para YCbCr, visto este ser o formato *standard* de entrada do compressor JPEG.

A funcionalidade *Entropy* apresenta duas funcionalidades alternativas: *Huffman* e *Arithmetic*. Embora a codificação *Arithmetic* seja superior em termos de rapidez e eficiência, a mais usual é a codificação *Huffman*. A justificação reside no facto de a codificação *Arithmetic* ser patenteada e haver restrições no seu uso.

A funcionalidade *Compressão* apresenta quatro funcionalidades alternativas: *Sequencial DCT*, *Sequencial Lossless*, *Progressive DCT* e *Hierarchical*. A funcionalidade *Sequencial DCT* representa o formato *baseline* do JPEG (o modo de compressão mais utilizado no formato JPEG). A funcionalidade *Sequencial Lossless* é utilizada em aplicações onde não existe a necessidade de obter imagens com grande qualidade. A funcionalidade *Progressive DCT* é semelhante ao modo *baseline* mas em vez de processar a informação da imagem de uma só vez, processa a imagem em vários *scans*, o que faz com que a imagem final tenha melhor resolução. Finalmente, a funcionalidade *Hierarchical* permite que a imagem comprimida contenha vários tipos de resolução no mesmo ficheiro JPEG.

A funcionalidade *Sampling* apresenta três funcionalidades alternativas: *Sampling 4:4:4*, *Sampling 4:2:2* e *Sampling 4:1:1*. A Figura 6.5 apresenta cada um deles.

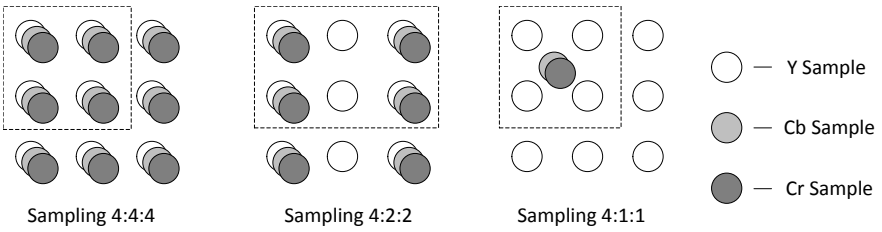


Figura 6.5: *Samplings* do formato JPEG.

Como o olho Humano é mais sensível à luz do que à cor numa imagem, a imagem a comprimir pode ter mais amostras da componente da luminância (luz) do que de cada uma das crominâncias (cores). Assim, no *Sampling 4:4:4* para cada componente de Y existe uma componente de Cb e Cr, no *Sampling 4:2:2* para grupos de duas componentes de Y na vertical existe apenas uma componente de Cb e Cr e no *Sampling 4:1:1* para grupos de duas componentes de Y na vertical e horizontal existe apenas uma componente de Cb e Cr.

Finalmente, a funcionalidade *DCT* é constituída por três funcionalidades alternativas: *Intel*, *AMD* e *ARM*. Cada uma delas representa a implementação da transformada discreta do co-seno em cada uma das plataformas suportadas. Isto é, para cada plataforma é implementada a DCT usando instruções *assembly* para aumentar o desempenho.

Diagrama de Funcionalidades de um Filtro de Saída Para representar o diagrama de funcionalidades de um filtro de saída é dado como exemplo o filtro de saída *Ficheiro*. O filtro de saída *Ficheiro* apresenta a variabilidade que existe no formato do ficheiro a gravar e localização. A Figura 6.6 apresenta o diagrama de funcionalidades do filtro de saída *Ficheiro*.

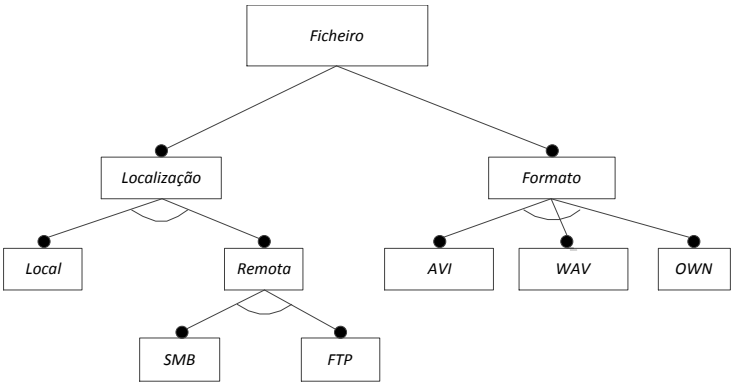


Figura 6.6: Diagrama de funcionalidades do filter de saída Ficheiro.

O nó raiz representa o conceito (*Ficheiro*) que é composto por duas funcionalidades obrigatórias: *Formato* e *Localização*. A funcionalidade formato especifica o formato de ficheiro utilizado para gravar os dados da *stream*. A funcionalidade *Localização* especifica se a gravação é local ou remota. Se a localização de gravação for remota, é necessário utilizar um protocolo de rede para a transmissão de dados com uma máquina remota.

Representação do Diagrama de Funcionalidades Para representar cada um dos diagramas de funcionalidades do sistema de videovigilância utilizou-se o FMP (*Feature Modeling Plugin*). O FMP é uma ferramenta de código aberto desenvolvida em Java pelo *Generative Software Development Lab* que permite editar e configurar diagramas de funcionalidades no Eclipse. A título de exemplo, a Figura 6.7 apresenta o diagrama de funcionalidades do *filter graph* no FMP.

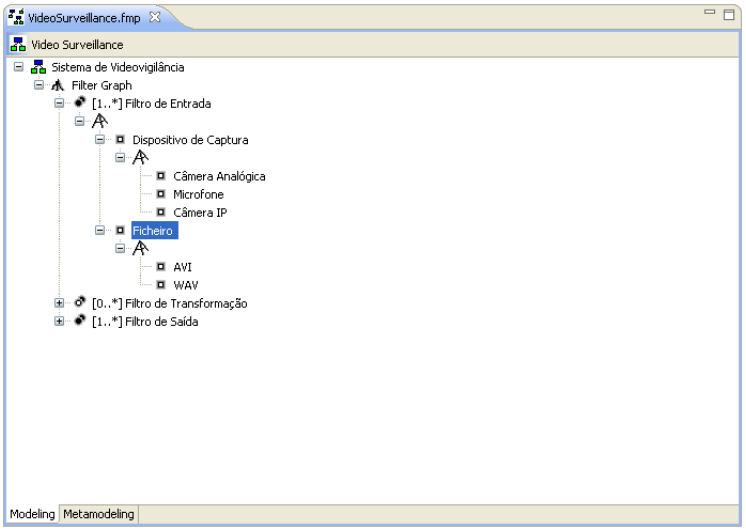


Figura 6.7: Diagrama de funcionalidades do *filter graph* no FMP.

Desenho do Domínio

O objetivo do desenho do domínio é definir qual a arquitetura que é utilizada na *framework* de videovigilância, identificar os componentes que serão implementados e especificar a API da *framework*. Para isso, são utilizados os resultados da modelação do domínio, essencialmente os diagramas de funcionalidades do *filter graph* e de cada um dos tipos de filtros que o compõem.

A *framework* de videovigilância tem como base uma arquitetura em *pipeline*, visto

que o *filter graph* não é mais do que um conjunto de filtros que estão ligados entre si de tal forma que o fluxo de dados segue o modelo de execução em *pipeline*.

Ao analisar os modelos de funcionalidade da fase da análise do domínio é possível identificar as funcionalidades base da *framework*. Cada uma dessas funcionalidades é implementada usando componentes altamente configuráveis e que definem a API da *framework*.

Domínio da Implementação

Depois de definida a arquitetura do sistema, o domínio da implementação foca-se no desenvolvimento de componentes configuráveis para dar suporte à implementação da arquitetura do domínio da videovigilância. A implementação da *framework* para a família de produtos da videovigilância é constituída por:

i. Um *template* artefacto *filter graph* que descreve o *pipeline* como um conjunto de filtros baseados na tecnologia de componentes desenvolvidos em C++ *template metaprogramming*;

ii. Um *template* artefacto filtro que descreve cada um dos tipos de filtros do sistema como um conjunto de pinos baseados na tecnologia de componentes desenvolvidos em C++ *template metaprogramming*;

iii. Um *template* artefacto pino desenvolvido em C++ *template metaprogramming* que permite que seja possível estabelecer a ligação entre dois filtros;

iv. Um conjunto de meta-expressões e condições de presença com correspondência direta aos *template* artefactos e os seus pontos de variabilidade, e expressos em termos de funcionalidades utilizadas para fazer o mapeamento das funcionalidades do modelo de funcionalidades e os *template* artefactos, e desta forma, dando significado a essas funcionalidades;

v. Um conjunto de regras implícitas que estabelecem e validam as ligações entre os filtros, que são internamente e implicitamente definidas como parte do *filter graph* através do *Intelligent Connector*. As regras são também desenvolvidas em C++ *template metaprogramming*.

Template* Artefacto *Filter Graph O *filter graph* é constituído por um conjunto de filtros que estão ligados entre si seguindo o modelo de execução em *pipe-*

line. Para obter uma determinada configuração do sistema é primeiro necessário identificar os filtros que compõem o *filter graph* e de seguida estabelecer a ligação entre eles. Essencialmente, a classe *template* que implementa o *filter graph*, CFilterGraph, varia no número de variáveis membro que definem cada um dos filtros e o tipo de dados dessas mesmas variáveis.

Para simplificar a compreensão da implementação da classe *template* CFilterGraph, será utilizado o exemplo da Figura 6.8.



Figura 6.8: Exemplo de uma configuração do *filter graph*.

O *filter graph* da Figura 6.8 é constituído por três filtros: o filtro de entrada CRTSPSource, o filtro de transformação CJPEGDecTransform e o filtro de saída CXVRenderer. Esta configuração representa um exemplo muito comum nos sistemas de videovigilância. O filtro de entrada CRTSPSource permite fazer a captura de imagens de uma câmara IP que suporta o protocolo RTSP. Como as imagens são enviadas no formato JPEG é necessário fazer a descompressão das mesmas usando o filtro CJPEGDecTransform para que estas possam posteriormente ser visualizadas no ecrã através do filtro CXVRenderer.

Diagrama de Classes A Figura 6.9 apresenta o diagrama de classes do *template* artefacto *filter graph*.

O *template* artefacto *filter graph* é representado pela classe *template* CFilterGraph. A classe *template* CFilterGraph recebe como parâmetro de entrada de *template* o vetor FilterTypes com os tipos de dados de cada filtro. Para que os tipos dos filtros do vetor sejam instanciados como variáveis membro da classe *template* CFilterGraph, esta herda da estrutura *template* TupleVector que devolve como resultado um tuplo com os tipos de dados definidos no vetor FilterTypes. A classe *template* tuplo definida na biblioteca *Boost Tuple* representa um objeto de dados constituído por um conjunto finito de elementos, os quais podem ser de diferentes tipos. No caso do *filter graph*, os elementos do tuplo são os tipos de dados dos filtros que compõem o *filter graph*:

```
template <typename FilterTypes>
class CFilterGraph
    : private TupleVector<FilterTypes>
{ ... };
```

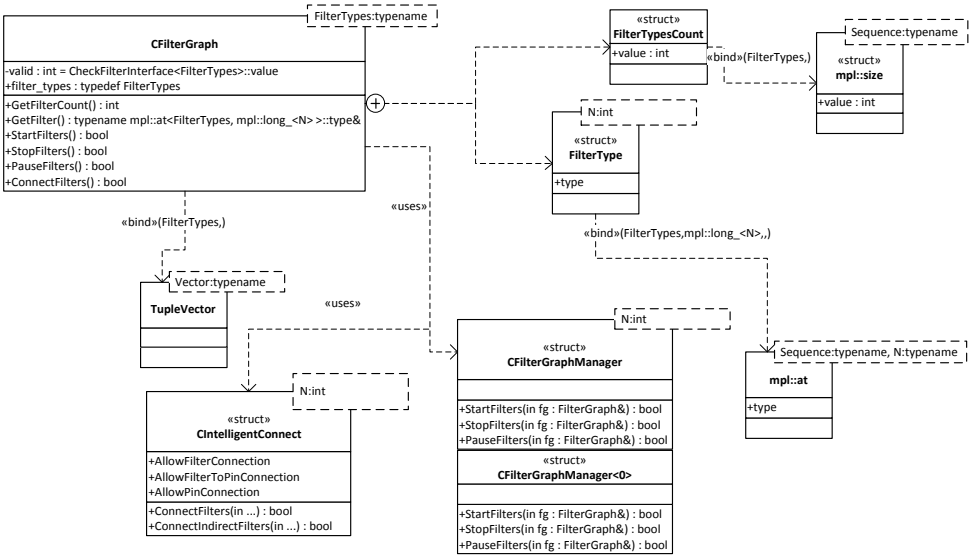


Figura 6.9: Diagrama de classes do *template* artefacto *filter graph*.

Para criar uma instância do *filter graph* da Figura 6.8 são necessárias as seguintes linhas de código:

```
typedef mpl::vector<CRTSPSource, CJPEGDecTransform, CXVRenderer> filter_types;
CFilterGraph<filter_types> fg;
```

Primeiro é necessário definir o vetor filter_types com os tipos dos filtros, para que este seja utilizado como parâmetro de entrada de *template* na classe *template* CFilterGraph ao definir a variável fg que representa o *filter graph*.

Acesso aos Filtros A classe *template* CFilterGraph implementa duas metafunções e dois métodos que permitem aceder à informação dos filtros que compõem o *filter graph*. As metafunções FilterTypesCount e FilterType permitem aceder à informação dos filtros em tempo de compilação. Os métodos GetFilterCount e GetFilter permitem aceder à informação dos filtros em tempo de execução.

A metafunção FilterTypesCount devolve o número de filtros que compõem o *filter graph* em tempo de compilação:

```
struct FilterTypesCount
    : public mpl::size<FilterTypes>
{};
```

O número de filtros é obtido usando a metafunção size da biblioteca *Boost MPL* que devolve o número de elementos do vetor FilterTypes.

A metafunção FilterType devolve o tipo de dados do filtro com índice N do *filter*

graph em tempo de compilação:

```
template <int N>
struct FilterType
    : public mpl::at<FilterTypes, mpl::long_<N> >
{};
```

Para obter o tipo de dados do filtro com o índice N, a metafunção recebe como parâmetro de entrada de *template* o índice do filtro, e devolve o tipo de filtro de índice N do vetor FilterTypes.

O método GetFilterCount devolve o número de filtros que compõem o *filter graph* em tempo de execução:

```
int GetFilterCount() const {
    return FilterTypesCount::value;
}
```

O número de filtros do *filter graph* é obtido utilizando a variável value da metafunção FilterTypesCount.

O método GetFilter devolve a referência do filtro com índice N do *filter graph* em tempo de execução:

```
template<int N>
typename FilterType<N>::type& GetFilter() {
    return get<N>();
}
```

Para obter a referência do filtro com o índice N, o método *template* recebe como parâmetro de entrada de *template* o índice do filtro, e devolve o filtro com o tipo de dados de índice N do vetor FilterTypes; usa a metafunção get da classe *template* tuple para devolver a referência do filtro pretendido.

Para aceder a cada um dos filtros do *filter graph* fg são necessárias as seguintes linhas de código:

```
CRTSPSource& sourceFilter = fg.GetFilter<0>();
CJPEGDecTransform& transformFilter = fg.GetFilter<1>();
CXVRenderer& rendererFilter = fg.GetFilter<2>();
```

Ligação entre os Filtros Depois de definidos os tipos de dados de cada um dos filtros que compõem o *filter graph*, é necessário estabelecer a ligação entre eles para obter uma configuração específica. O método *template* ConnectFilters da classe *template* CFilterGraph estabelece as ligações entre os filtros:

```
template <typename FilterConnections>
bool ConnectFilters() {
   return CIntelligentConnect<mpl::size<FilterConnections>::value>::template
         ConnectFilters<FilterTypes, FilterConnections,CFilterGraph>(*this);
}
```

O método *template* ConnectFilters recebe como parâmetro de entrada de *template* o vetor FilterConnections que contém a informação das ligações entre os filtros. Cada item do vetor FilterConnections é constituído por um par ordenado que contém os índices dos filtros que serão ligados. Para representar o par ordenado é utilizada a estrutura *template* pair da biblioteca *Boost MPL*.

A classe *template* CIntelligentConnect utilizada no método ConnectFilters, recebe como parâmetro de entrada de *template* o número de filtros definidos no vetor FilterTypes. Por sua vez, o método *template* ConnectFilters da classe *template* CIntelligentConnect recebe como parâmetro de entrada de *template* três parâmetros: o vetor FilterTypes com os tipos dos filtros, o vetor FilterConnections com os pares ordenados das ligações entre os filtros e o tipo de dados CFilterGraph que define o tipo de dados do *filter graph*. Como parâmetro de entrada, o método recebe a referência para o apontador this da classe CFilterGraph de forma a que seja possível aceder aos filtros que compõem o *filter graph*. As seguintes linhas de código permitem criar as ligações do *filter graph* da Figura 6.8:

```
typedef mpl::vector<CRTSPSource, CJPEGDecTransform, CXVRenderer> filter_types;
CFilterGraph<filter_types> fg;

typedef mpl::pair<mpl::int_<0>, mpl::int_<1> > connection1;
typedef mpl::pair<mpl::int_<1>, mpl::int_<2> > connection2;
typedef mpl::vector<connection1, connection2> filter_connections;
fg<filter_connections>.ConnectFilters();
```

Primeiro é definido o par ordenado connection1 que representa a ligação entre o filtro de índice 0 (CRTSPSource) e o filtro de índice 1 (CJPEGDecTransform). De seguida é definido o par ordenado connection2 que representa a ligação entre o filtro de índice 1 (CJPEGDecTransform) e o filtro de índice 2 (CXVRenderer). Depois é definido o vetor filter_connections com os pares ordenados connections1 e connections2. Por fim, o método ConnectFilters do *filter graph* fg recebe como parâmetro de entrada de *template* o vetor filter_connection.

Controlo do Fluxo de Execução As funções que controlam o fluxo de execução e o estado do *filter graph* são implementadas nos métodos: StartFilters, StopFilters e PauseFilters. O método StartFilters inicia o fluxo de execução do *filter graph*:

```
bool StartFilters() {
    return CFilterGraphManager<FilterTypesCount::value>::template
        StartFilters<FilterTypes, CFilterGraph>(*this);
}
```

Ao iniciar o fluxo de execução do *filter graph* é iniciado o fluxo de execução em cada um dos filtros que compõem o *filter graph*. Para iniciar o fluxo de execução em cada um dos filtros, é executado o método *template* StartFilters da estrutura *template* CFilterGraphManager. A estrutura *template* CFilterGraphManager recebe como parâmetro de entrada de *template* o número de filtros definidos no vetor FilterTypes. Por sua vez, o método *template* StartFilters recebe como parâmetro de entrada de *template* o vetor FilterTypes com os tipos dos filtros e o tipo de dados do *filter graph*, CFilterGraph. Como parâmetro de entrada, o método recebe a referência para o apontador this da classe CFilterGraph de forma a que seja possível aceder aos filtros que compõem o *filter graph*.

O método StopFilters termina o fluxo de execução do *filter graph* que, por sua vez, termina o fluxo de execução de cada um dos filtros do *filter graph*:

```
bool StopFilters() {
    return CFilterGraphManager<FilterTypesCount::value>::value>::template
        StopFilters<FilterTypes, CFilterGraph>(*this);
}
```

Finalmente, o método PauseFilters faz uma pausa no fluxo de execução do *filter graph*, e consequentemente em cada um dos filtros do *filter graph*:

```
bool PauseFilters() {
    return CFilterGraphManager<FilterTypesCount::value>::value>::template
        PauseFilters<FilterTypes, CFilterGraph>(*this);
}
```

Os métodos StopFilters e PauseFilters não foram explicados em detalhe, porque funcionam de modo análogo ao método StartFilters. As linhas de código que se seguem permitem iniciar o fluxo de execução do *filter graph* da Figura 6.8 e termina o fluxo de execução ao fim de cinco segundos:

```
typedef mpl::vector<CRTSPSource, CJPEGDecTransform, CXVRenderer> filter_types;
CFilterGraph<filter_types> fg;

typedef mpl::pair<mpl::int_<0>, mpl::int_<1> > connection1;
typedef mpl::pair<mpl::int_<1>, mpl::int_<2> > connection2;
typedef mpl::vector<connection1, connection2> filter_connections;
fg<filter_connections>.ConnectFilters();

fg.StartFilters();
CUtils::Sleep(5);
fg.StopFilters();
```

Filter Graph Manager O *filter graph manager* é utilizado pelo *filter graph* para controlar o fluxo de execução de cada um dos filtros que compõem o *filter graph*. Tal como ilustra a Figura 6.9, a estrutura *template* CFilterGraphManager implementa uma *template* genérica e uma *template* especializada. A *template* genérica inicia o processo recursivo de execução de um determinado método em cada um dos filtros do *filter graph*. Como o processo é recursivo, é necessário implementar uma condição de paragem que é implementada pela *template* especializada quando o índice N da execução do processo recursivo é 0, isto é, quando o número de filtros a processar é nulo.

Método *StartFilters* Para iniciar o fluxo de execução de cada filtro é definido na *template* genérica da classe *template* CFilterGraphManager o método *template* StartFilters:

```
template<int N>
struct CFilterGraphManager {
    ...
    template<typename FilterTypes, typename FilterGraph>
    inline static bool StartFilters(FilterGraph& fg) {
        return (CFilterGraphManager<N−1>::template
            StartFilters<FilterTypes, FilterGraph>(fg) &
                fg.GetFilter<N−1>().Start());
    }
    ...
};
```

O método recebe como parâmetros de entrada de *template* o vetor FilterTypes com os tipos de dados dos filtros e o tipo de dados FilterGraph que representa o tipo de dados da classe *filter graph*. Como parâmetro de entrada do método, recebe uma referência para o *filter graph* para aceder a cada um dos filtros que o compõem.

Em cada processo iterativo o resultado do método *template* StartFilters é obtido aplicando a operação lógica *and* entre o resultado do mesmo método chamado de forma recursiva com o parâmetro de entrada N-1 e o resultado do método Start do filtro com índice N-1 do *filter graph*. Como o processo é recursivo, implica que seja implementada uma *template* especializada quando o índice N-1 é 0, isto é, o número de filtros a processar é nulo:

```
template<>
struct CFilterGraphManager<0> {
    ...
    template<typename FilterTypes, typename FilterGraph>
    inline static bool StartFilters(FilterGraph& fg) {
        return true;
    }
```

```
    ...
};
```

Como o método *template* StartFilters da *template* especializada é *inline*, o compilador injeta o código no ato de instanciação do método em vez de chamar o método. A Figura 6.10 apresenta o processo iterativo da método *template* StartFilters do exemplo do *filter graph* da Figura 6.8.

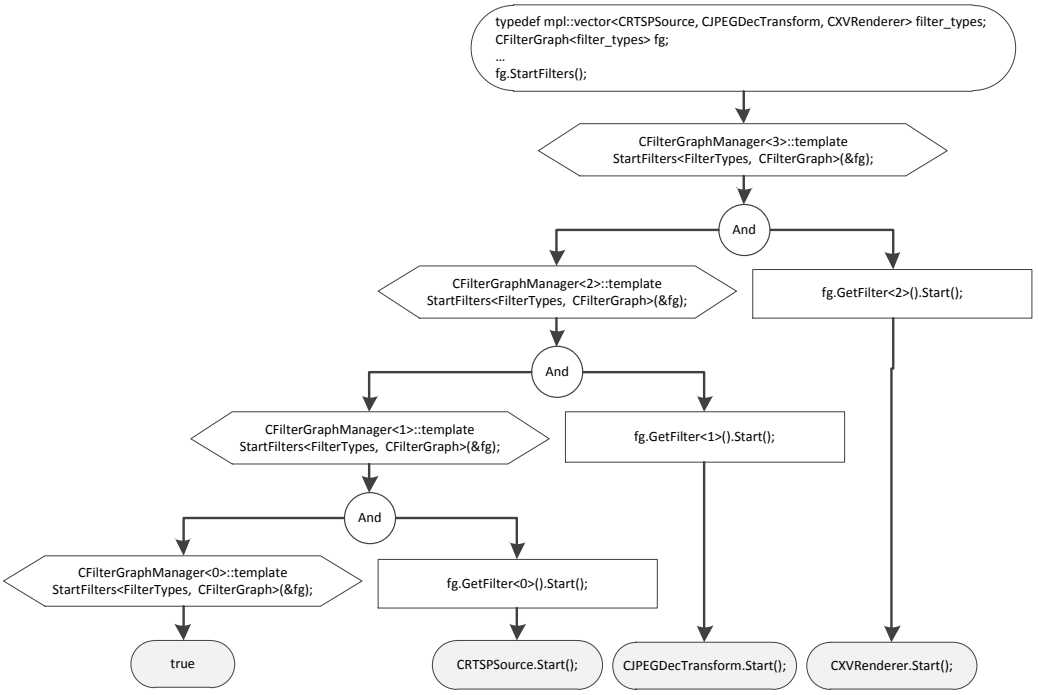


Figura 6.10: Processo iterativo do método StartFilters do exemplo do *filter graph* da Figura 6.8.

Método *StopFilters* Para terminar o fluxo de execução de cada filtro, é definido na *template* genérica da classe *template* CFilterGraphManager o método *template* StopFilters:

```
template<int N>
struct CFilterGraphManager {
    ...
    template<typename FilterTypes, typename FilterGraph>
    inline static bool StopFilters(FilterGraph& fg) {
        return (CFilterGraphManager<N−1>::template
            StopFilters<FilterTypes, FilterGraph>(fg) &
                fg.GetFilter<N−1>().Stop());
    }
    ...
};
```

A condição de paragem do método *template* StopFilters é definida na *template* especializada da classe *template* CFilterGraphManager:

```
template<>
struct CFilterGraphManager<0> {
    ...
    template<typename FilterTypes, typename FilterGraph>
    inline static bool StopFilters(FilterGraph& fg) {
        return true;
    }
    ...
};
```

Método *PauseFilters* Para fazer pausa no fluxo de execução de cada filtro, é definido na *template* genérica da classe *template* CFilterGraphManager o método *template* PauseFilters:

```
template<int N>
struct CFilterGraphManager {
    ...
    template<typename FilterTypes, typename FilterGraph>
    inline static bool PauseFilters(FilterGraph& fg) {
        return (CFilterGraphManager<N-1>::template
            PauseFilters<FilterTypes, FilterGraph>(fg) &
                fg.GetFilter<N-1>().Pause());
    }
    ...
};
```

A condição de paragem do método *template* PauseFilters é definida na *template* especializada da classe *template* CFilterGraphManager:

```
template<>
struct CFilterGraphManager<0> {
    ...
    template<typename FilterTypes, typename FilterGraph>
    inline static bool PauseFilters(FilterGraph& fg) {
        return true;
    }
    ...
};
```

Os métodos *template* StopFilters e PauseFilters têm o mesmo comportamento ao nível do funcionamento que o método *template* StartFilter. O processo iterativo dos dois métodos *template* é igual ao processo iterativo do método *template* StartFilters, mas em vez de chamar o método *template* Start de cada filtro, é chamado o método Stop e Pause, respetivamente.

***Template* Artefacto Filtro** Os filtros são os componentes elementares do *filter graph*. É através da combinação dos vários filtros que é possível obter várias configurações do sistema. Para obter uma determinada configuração, é necessário que os filtros se possam ligar uns aos outros. A ligação entre filtros só é possível através de pinos, que podem ser de entrada ou de saída, que são ligados entre si se o tipo de *media* de áudio/vídeo suportado pelos dois for compatível. Um filtro é constituído por um conjunto de pinos que permitem estabelecer ligações com pinos de outros filtros do mesmo *filter graph*. Essencialmente, o *template* artefacto filtro varia no número de variáveis membro que definem cada um dos pinos e o tipo de dados dessas mesmas variáveis.

A Figura 6.11 apresenta o exemplo da Figura 6.8 onde é possível ver os pinos que compõem cada um dos filtros. O filtro de entrada CRTSPSource contém apenas o pino de saída P1.0 do tipo CRTSPOutputPin. O filtro de transformação CJPEGDecTransform é constituído por dois pinos: o pino de entrada P2.0 do tipo CJPEGDecInputPin e o pino de saída P2.1 do tipo CJPEGDecOutputPin. Finalmente, o filtro de saída CXVRenderer contém apenas o pino de entrada P3.0 do tipo CXVInputPin.

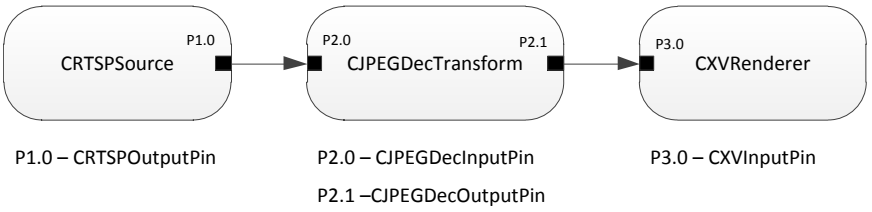


Figura 6.11: Exemplo da Figura 6.8 com os pinos.

A ligação entre o pino de saída P1.0 do filtro de entrada CRTSPSource e o pino de entrada P2.0 do filtro de transformação CJPEGDecTransform é possível se e só se os dois pinos suportarem o mesmo tipo de *media* de vídeo, neste caso o tipo de *media* de vídeo JPEG_MEDIA_TYPE. De igual modo, a ligação entre o pino de saída P2.1 do filtro de transformação CJPEGDecTransform e o pino de entrada P3.0 do filtro de saída CXVRenderer é possível, porque os dois suportam o tipo de *media* de vídeo YUV420_MEDIA_TYPE.

Tipo de Filtros Tal como acontece no *DirectShow*, a *framework* generativa de videovigilância implementa três tipos de filtros: filtro de entrada, filtro de transformação e filtro de saída. Para definir os vários tipos de filtros, são utilizadas estruturas que permitem identificar cada um dos filtros em função do seu tipo

usando *template metaprogramming*. As linhas de código seguintes definem os tipos de filtros suportados pela *framework*:

```
// estrutura que representa um filtro de entrada
struct source_filter_t {};

// estrutura que representa um filtro de transformação
struct transform_filter_t {};

// estrutura que representa um filtro de saída
struct renderer_filter_t {};
```

Para verificar qual o tipo de um determinado filtro, foram definidas três metafunções. A metafunção is_source_filter verifica se um filtro é do tipo filtro de entrada:

```
template <typename FilterType>
struct is_source_filter
    : public is_same<typename FilterType::type, source_filter_t>
{};
```

A metafunção is_transform_filter verifica se um filtro é do tipo filtro de transformação:

```
template <typename FilterType>
struct is_transform_filter
    : public is_same<typename FilterType::type, transform_filter_t>
{};
```

Finalmente, a metafunção is_renderer_filter verifica se um filtro é do tipo filtro de saída:

```
template <typename FilterType>
struct is_renderer_filter
    : public is_same<typename FilterType::type, renderer_filter_t>
{};
```

As metafunções apresentadas utilizam o mesmo padrão de implementação e o mesmo modo de funcionamento. Todas elas recebem como parâmetro de entrada de *template* o tipo de dados FilterType que representa o tipo de dados do filtro a testar. Para verificar qual o tipo de filtro a testar, FilterType, todas as metafunções herdam da metafunção is_same da biblioteca *Boost*. A metafunção is_same devolve verdade se os dois tipos de dados passados como parâmetros de entrada de *template* forem do mesmo tipo.

As linhas de código seguintes verificam se o filtro CRTSPSource é um filtro de transformação:

```
bool value = is_transform_filter<CRTSPSource>::value;
```

O resultado da metafunção is_transform_filter é falso, porque o filtro CRTSPSource é um filtro de entrada.

Diagrama de Classes A Figura 6.12 apresenta o diagrama de classes do *template* artefacto filtro.

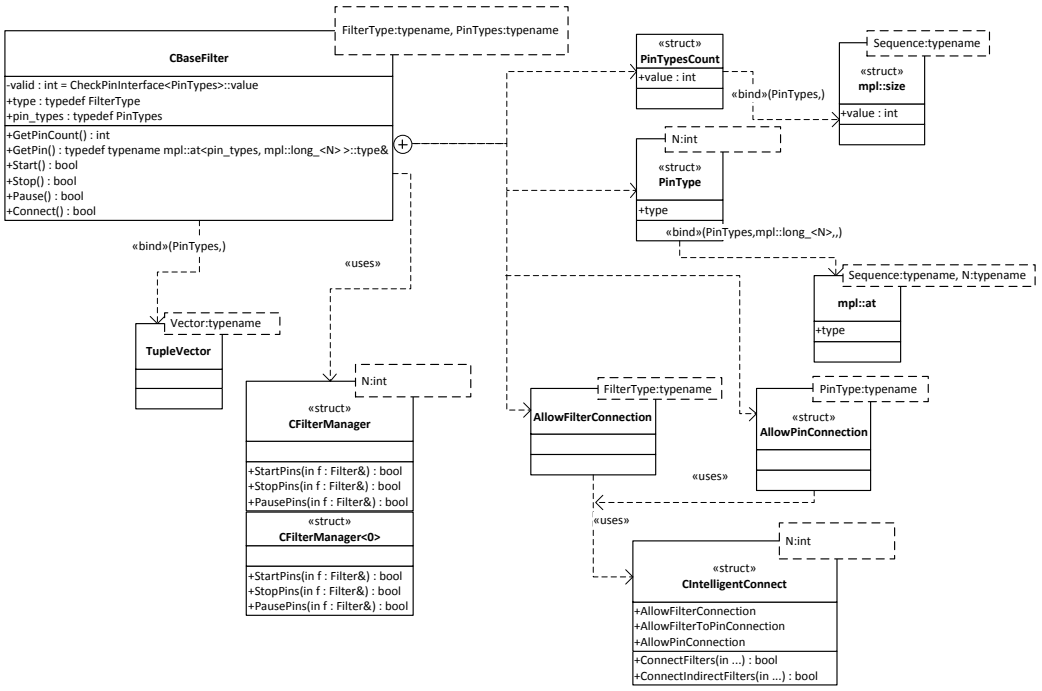


Figura 6.12: Diagrama de classes do *template* artefacto filtro.

O *template* artefacto filtro é representado pela classe *template* CBaseFilter. Todos os filtros implementados na *framework* generativa de videovigilância herdam da classe *template* CBaseFilter. A classe *template* recebe dois parâmetros de entrada de *template*: o tipo de dados FilterType e o vetor PinTypes. O tipo de dados FilterType define o tipo de filtro e o vetor PinTypes define o tipo de dados de cada pino. Para que os tipos dos pinos do vetor sejam instanciados como variáveis membro da classe *template* CBaseFilter, esta herda da estrutura *template* TupleVector que devolve como resultado um tuplo com os tipos de dados definidos no vetor PinTypes:

```
template <typename FilterType, typename PinTypes>
class CBaseFilter
    : private TupleVector<PinTypes>
{ ... };
```

O código seguinte apresenta a definição da classe que implementa o filtro de transformação CJPEGDecTransform, que é constituído por dois pinos: o pino de entrada CJPEGDecInputPin e o pino de saída CJPEGDecOutputPin.

```
typedef mpl::vector<CJPEGDecInputPin, CJPEGDecOutputPin> pin_types;

class CJPEGDecTransform
    : public CBaseFilter<transform_filter_t, pin_types>
{
```

```
public:
    CJPEGDecTransform();
    ~CJPEGDecTransform();
    ...
};
```

Numa primeira fase é definido o vetor pin_types que contém a informação dos dois tipos de pinos que compõem o filtro de transformação CJPEGDecTransform (o tipo CJPEGDecInputPin e o tipo CJPEGDecOutputPin). A classe que implementa o filtro de transformação CJPEGDecTransform herda da classe *template* CBaseFilter. A classe *template* CBaseFilter recebe como parâmetros de entrada o tipo de filtro (transform_filter_t) e o vetor com os tipos dos pinos (pin_types).

***Interface* Estática do *Template* Artefacto Filtro** No *DirectShow*, todos os filtros herdam da *interface* IBaseFilter. Basicamente, a maioria dos métodos virtuais definidos na *interface* IBaseFilter são implementados pela classe abstrata CBaseFilter. O resto dos métodos virtuais da *interface* IBaseFilter que não são implementados pela classe CBaseFilter, são implementados pelas classes CSource, CTransformFilter e CBaseRenderer que herdam da classe CBaseFilter, caso o filtro a implementar seja um filtro de entrada, um filtro de transformação ou um filtro de saída, respetivamente. Como é usado polimorfismo dinâmico, a variabilidade do sistema é resolvida apenas em tempo de execução.

Ao contrário do *DirectShow*, em vez de se definirem classes abstratas que implementam cada um dos tipos de filtros (CSource, CTransformFilter e CBaseRenderer), todos os filtros, independentemente do seu tipo, seguem a mesma base de implementação, o que permite representar o artefacto filtro de uma forma mais genérica e flexível. Desta forma, é mais simples fazer o *refactoring* do código *legacy* de bibliotecas externas que implementam as funcionalidades de alguns filtros.

Como a variabilidade do sistema é resolvida em tempo de compilação, foi definida uma *interface* estática que todos os filtros necessitam de implementar. Para definir a *interface* estática dos filtros, utilizou-se a *Boost Concept Checking Library* (BCCL), que define a funcionalidade *concept*, que verifica se os parâmetros de entrada de uma determinada *template* satisfazem determinados requisitos por exemplo, a definição de determinadas variáveis e métodos.

Para criar um novo filtro, é necessário que o novo filtro herde da classe *template* CBaseFilter, a qual define a seguinte *interface* estática:

i. O tipo de dados type, que indica qual o tipo de filtro a implementar (filtro de entrada, filtro de transformação ou filtro de saída);

ii. O tipo de dados pin_types, que define um vetor com o tipo de pinos que compõem o filtro;

iii. Os métodos Start, Stop e Pause, que controlam o estado e o fluxo de execução do filtro;

iv. O método Connect, que permite ligar o filtro a um outro filtro do mesmo *filter graph*.

O código seguinte cria o *concept* FilterConcept, que define a *interface* estática do *template* artefacto filtro:

```
template <typename F>
struct FilterConcept {
    void constraints() {
        typedef typename F::type type; // define filter type
        typedef typename F::pin_types pin_types; // define filter pins
        f.Start(); // F needs to implement Start method
        f.Stop(); // F needs to implement Stop method
        f.Pause(); // F needs to implement Pause method
        f.Connect(F&); // F needs to implement Connect method
    }
    F f;
};
```

Acesso aos Pinos Tal como na classe *template* CFilterGraph, a classe *template* CBaseFilter implementa duas metafunções e dois métodos que permitem aceder à informação dos pinos que compõem um determinado filtro. As metafunções PinTypesCount e PinType permitem aceder à informação dos pinos em tempo de compilação. Os métodos GetPinCount e GetPin permitem aceder à informação dos pinos em tempo de execução.

A metafunção PinTypesCount devolve o número de pinos do filtros em tempo de compilação:

```
struct PinTypesCount
    : public mpl::size<PinTypes>
{};
```

O número de pinos é obtido usando a metafunção size da biblioteca *Boost MPL*, que devolve o número de elementos do vetor PinTypes.

O metafunção PinType devolve o tipo de dados do pino com índice N do filtro em tempo de compilação:

```
template <int N>
struct PinType
   : public mpl::at<PinTypes, mpl::long_<N> >
{};
```

Para obter o tipo de dados do pino com o índice N, a metafunção recebe como parâmetro de entrada de *template* o índice do pino, e devolve o tipo de pino de índice N do vetor PinTypes.

O método GetPinCount devolve o número de pinos que compõem o filtro em tempo de execução:

```
int GetPinCount() const {
   return PinTypesCount::value;
}
```

O número de pinos é obtido usando a variável value da metafunção PinTypesCount.

O método GetPin devolve a referência do pino com índice N do filtro em tempo de execução:

```
template<int N>
typename PinType::type& GetPin() {
   return get<N>();
}
```

Para obter a referência do pino com o índice N, o método recebe como parâmetro de entrada de *template* o índice do pino, que devolve o pino com o tipo de dados de índice N do vetor PinTypes; usa a metafunção get da classe *template* tuple para devolver a referência do pino pretendido.

As linhas de código seguintes permitem aceder a todos os pinos dos filtros que compõem o *filter graph* fg:

```
CRTSPOutputPin& pin1 = fg.GetFilter<0>().GetPin<0>();
CJPEGDecInputPin& pin2 = fg.GetFilter<1>().GetPin<0>();
CJPEGDecOutputPin& pin3 = fg.GetFilter<1>().GetPin<1>();
CXVInputPin& pin4 = fg.GetFilter<2>().GetPin<0>();
```

Ligação entre dois Filtros Para estabelecer a ligação entre dois filtros, é utilizado o método *template* Connect da classe *template* CBaseFilter:

```
template<typename Filter>
bool Connect(Filter& filter) {
   // check if the connection is possible
```

```
    static_assert(AllowFilterConnection<Filter>::value,
                  "It is not possible to connect the filters!");
    static const int output = AllowFilterConnection<Filter>::output;
    static const int input = AllowFilterConnection<Filter>::input;
    return this->GetPin<output>().Connect(
                  &filter.GetPin<input>(), filter.GetPin<input>().Process);
}
```

O método *template* Connect recebe o parâmetro de entrada de *template* Filter, que define o tipo de dados do filtro ao qual se vai estabelecer a ligação, e como parâmetro de entrada do método, recebe a referência ao filtro em questão. Numa primeira fase, o método verifica se é possível estabelecer a ligação entre os dois filtros usando a metafunção AllowFilterConnection. Caso a ligação não seja possível, a função static_assert apresenta uma mensagem de erro no ato de compilação a informar que não é possível estabelecer a ligação entre os dois filtros.

Se a ligação entre os dois filtros for possível, a metafunção AllowFilterConnection devolve o índice do pino do filtro que estabelece a ligação, output, e o índice do pino ao qual se vai estabelecer a ligação, input. Por fim, o método acede aos pinos de cada filtro através do método GetPin da classe *template* CBaseFilter e executa o método Connect do pino que estabelece a ligação. O método Connect da classe CBasePin que será apresentada mais à frente neste capítulo recebe como parâmetro de entrada o apontador para o pino que recebe a ligação e o método Process do mesmo pino.

A metafunção AllowFilterConnection da estrutura CIntelligentConnect, serve de base de implementação da metafunção AllowFilterConnection, que verifica se a ligação entre os dois filtros, CBaseFilter e FilterType, é possível. Caso a ligação seja possível, é devolvido o índice do pino do filtro que inicia a ligação output e o índice do pino do filtro que recebe a ligação input. Os índices dos pinos permitem aceder a cada um dos pinos através do método *template* GetPin da classe *template* CBaseFilter.

```
template<typename FilterType>
struct AllowFilterConnection
    : CIntelligentConnect<PinTypesCount::value>::template
          AllowFilterConnection<CBaseFilter, FilterType>
{};
```

As linhas de código seguintes estabelecem as ligações entre os filtros que compõem o *filter graph* da Figura 6.11:

```
// CRTSPSource -> CJPEGDecTransform
fg.GetFilter<0>().Connect(fg.GetFilter<1>());
// CJPEGDecTransform -> CXVRenderer
fg.GetFilter<1>().Connect(fg.GetFilter<2>());
```

Controlo do Fluxo de Execução As funções que controlam o fluxo de execução e o estado do filtro são implementadas nos métodos *template* da classe *template* CBaseFilter: Start, Stop e Pause.

O método Start inicia o fluxo de execução do filtro:

```
bool Start() {
    return CFilterManager<PinTypesCount::value>::template
        StartPins<PinTypes, CBaseFilter>(*this);
}
```

Ao iniciar o fluxo de execução do filtro, é iniciado o fluxo de execução em cada um dos pinos que compõem o filtro. Para iniciar o fluxo de execução em cada um dos pinos, é executado o método *template* StartPins da classe *template* CFilterManager. A classe *template* CFilterManager recebe como parâmetro de entrada o número de pinos definidos no vetor PinTypes. Por sua vez, o método *template* StartPins recebe como parâmetro de entrada o vetor PinTypes com os tipos dos pinos e o tipo de dados do filtro, CBaseFilter. Como parâmetro de entrada, o método recebe a referência para o apontador this da classe CBaseFilter de forma a que seja possível aceder aos pinos que compõem o filtro.

O método Stop termina o fluxo de execução do filtro que, por sua vez, termina o fluxo de execução de cada um dos pinos do filtro:

```
bool Stop() {
    return CFilterManager<PinTypesCount::value>::template
        StopPins<PinTypes, CBaseFilter>(*this);
  }
```

Finalmente, o método Pause faz uma pausa no fluxo de execução do filtro e, consequentemente, em cada um dos pinos do filtro.

```
bool Pause() {
    return CFilterManager<PinTypesCount::value>::template
        PausePins<PinTypes, CBaseFilter>(*this);
}
```

Os métodos Stop e Pause não foram explicados em detalhe, porque funcionam de forma análoga ao método Start.

Para iniciar o fluxo de execução do *filter graph* da Figura 6.11, são necessárias as seguintes linhas de código:

```
fg.GetFilter<0>().Start(); // CRTSPSource
fg.GetFilter<1>().Start(); // CJPEGDecTransform
fg.GetFilter<2>().Start(); // CXVRenderer
```

Filter Manager O *filter manager* é usado pela classe *template* CBaseFilter para controlar o fluxo de execução de cada um dos pinos que compõem o filtro. Tal como ilustra a Figura 6.12, a classe *template* CFilterManager implementa uma *template* genérica e uma *template* especializada. A *template* genérica inicia o processo recursivo de execução de um determinado método em cada um dos pinos do filtro. Como o processo é recursivo, é necessário implementar uma condição de paragem que é fornecida pela *template* especializada quando o índice N da execução do processo recursivo é 0, isto é, quando o número de pinos a processar é nulo.

Método *StartPins* Para iniciar o fluxo de execução de cada pino, é definido na *template* genérica da classe *template* CFilterManager o método *template* StartPins:

```
template<int N>
struct CFilterManager {
    ...
    template<typename PinTypes, typename Filter>
    inline static bool StartPins(Filter& f) {
        return (CFilterManager<N−1>::template
            StartPins<PinTypes, Filter>(f) &
                f.GetPin<N−1>().Start());
    }
    ...
};
```

O método recebe como parâmetros de entrada de *template* o vetor PinTypes com os tipos de dados dos pinos e o tipo de dados Filter, que representa o tipo de dados da classe filtro. Como parâmetro de entrada do método, recebe uma referência para o filtro para aceder a cada um dos pinos que o compõem. A condição de paragem do método *template* StartPins é definida na *template* especializada da classe *template* CFilterManager:

```
template<>
struct CFilterManager<0> {
    ...
    template<typename PinTypes, typename Filter>
    inline static bool StartPins(Filter& f) {
        return true;
    }
    ...
};
```

Método *StopPins* Para terminar o fluxo de execução de cada pino, é definido na *template* genérica da estrutura *template* CFilterManager o método *template* StopPins:

```
template<int N>
struct CFilterManager {
    ...
    template<typename PinTypes, typename Filter>
    inline static bool StopPins(Filter& f) {
        return (CFilterManager<N−1>::template
            StopPins<PinTypes, Filter>(f) &
                f.GetPin<N−1>().Stop());
    }
    ...
};
```

A condição de paragem do método *template* StopPins é definido na *template* especializada da classe *template* CFilterManager:

```
template<>
struct CFilterManager<0> {
    ...
    template<typename PinTypes, typename Filter>
    inline static bool StopPins(Filter& f) {
        return true;
    }
    ...
};
```

Método *PausePins* Para fazer pausa no fluxo de execução de cada pino, é definido na *template* genérica da classe *template* CFilterManager o método *template* PausePins:

```
template<int N>
struct CFilterManager {
    ...
    template<typename PinTypes, typename Filter>
    inline static bool PausePins(Filter& f) {
        return (CFilterManager<N−1>::template
            PausePins<PinTypes, Filter>(f) &
                f.GetPin<N−1>().Pause());
    }
    ...
};
```

A condição de paragem do método *template* PausePins é definido na *template* especializada da classe *template* CFilterManager:

```
template<>
struct CFilterManager<0> {
    template<typename PinTypes, typename Filter>
    inline static bool PausePins(Filter& f) {
        return true;
    }
    ...
};
```

***Template* Artefacto Pino** Os pinos representam os pontos de ligação entre os filtros. A ligação entre dois pinos só é possível, se forem validadas as seguintes regras: (i) os pinos têm que pertencer a diferentes filtros, (ii) os filtros têm que pertencer ao mesmo *filter graph*, (iii) os pinos têm que ser de diferentes tipos e (iv) os pinos têm que suportar o mesmo tipo de *media* de áudio/vídeo.

Durante o processo de ligação de dois pinos, é possível identificar as sinergias que possam existir entre ambos. Assim, é possível otimizar o código da cada um dos filtros, em função da ligação pretendida. Isto é, ao estabelecer a ligação entre dois pinos de diferentes filtros, é possível identificar as funcionalidades dos filtros necessárias para que a comunicação de dados entre os dois pinos seja possível. Logo, é possível compilar apenas o código necessário de cada um dos pinos e as funcionalidades dos filtros que dão suporte a cada um deles.

O pinos são também responsáveis por criar uma nova *thread* de processamento, que cria um fluxo de execução independente da *thread* principal do sistema de videovigilância. Como os filtros de entrada são a fonte de dados para os restantes filtros do *filter graph*, são responsáveis por criar uma nova *thread* de processamento para a aquisição de dados e envio dos mesmos aos restantes filtros.

A figura 6.13 apresenta os tipos de *media* de áudio/vídeo suportados por cada um dos pinos que compõem os filtros do *filter graph* da figura 6.8:

1. O pino de saída CRTSPOutputPin do filtro de entrada CRTSPSource, suporta dois tipos de *media* de vídeo: o tipo JPEG_MEDIA_TYPE (MT1.0) e o tipo H264_MEDIA_TYPE (MT1.1);

2. O pino de entrada CJPEGDecInputPin do filtro de transformação CJPEGDecTransform, apenas suporta o tipo de *media* de vídeo JPEG_MEDIA_TYPE (MT1.0);

3. O pino de saída CJPEGDecOutputPin do filtro de transformação CJPEGDecTransform, suporta dois tipos de *media* de vídeo: o tipo YUV420_MEDIA_TYPE (MT2.0) e o tipo RGB24_MEDIA_TYPE (MT2.1);

4. O pino de entrada CXVInputPin do filtro de saída CXVRenderer, apenas suporta um tipo de *media* de vídeo, o tipo YUV420_MEDIA_TYPE (MT1.0).

A ligação entre o pino de saída CRTSPOutputPin do filtro de entrada CRTSPSource e o pino de entrada CJPEGDecInputPin do filtro de transformação CJPEGDecTransform, é possível porque ambos suportam o tipo de *media* de vídeo JPEG_MEDIA_TYPE. Por sua vez, a ligação entre o pino de saída CJPEGDecOutputPin do filtro de transfor-

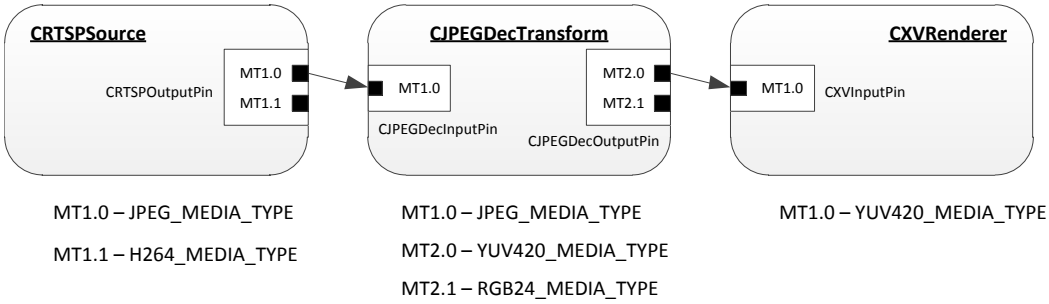


Figura 6.13: Tipos de *media* de áudio/vídeo suportados por cada um dos pinos que compõem os filtros do *filter graph*.

mação CJPEGDecTransform e o pino de entrada CXVInputPin do filtro de saída CXVRenderer, é possível porque os dois suportam o tipo de *media* de vídeo YUV420_MEDIA_TYPE.

Tipo de Pinos A *framework* generativa de videovigilância implementa dois tipos de pinos: pino de entrada e pino de saída. Para definir os dois tipos de pinos são utilizadas estruturas que permitem identificar cada um dos pinos em função do seu tipo usando *template metaprogramming*. As linhas de código seguintes definem os tipos de pinos suportados pela *framework*:

```
// estrutura que representa um pino de entrada
struct input_pin_t {};

// estrutura que representa um pino de saída
struct output_pin_t {};
```

Para verificar o tipo de pino de um determinado pino foram definidas duas metafunções. A metafunção is_input_pin verifica se um pino é do tipo pino de entrada:

```
template <typename PinType>
struct is_input_pin
    : public is_same<typename PinType::type, input_pin_t>
{};
```

A metafunção is_output_pin verifica se um pino é do tipo pino de saída:

```
template <typename PinType>
struct is_output_pin
    : public is_same<typename PinType::type, output_pin_t>
{};
```

As duas metafunções recebem como parâmetro de entrada de *template* o tipo de dados PinType que representa o tipo de dados do pino a testar. Para verificar qual o tipo de pino a testar, PinType, todas as metafunções herdam da metafunção is_same da biblioteca *Boost*.

As linhas de código seguintes verificam se o pino CRTSPOutputPin é um pino de saída:

```
bool value = is_output_pin<CRTSPOutputPin>::value;
```

O resultado da metafunção is_output_pin é verdade, porque o pino CRTSPOutputPin é um pino de saída.

Diagrama de Classes A Figura 6.14 apresenta o diagrama de classes do *template* artefacto pino.

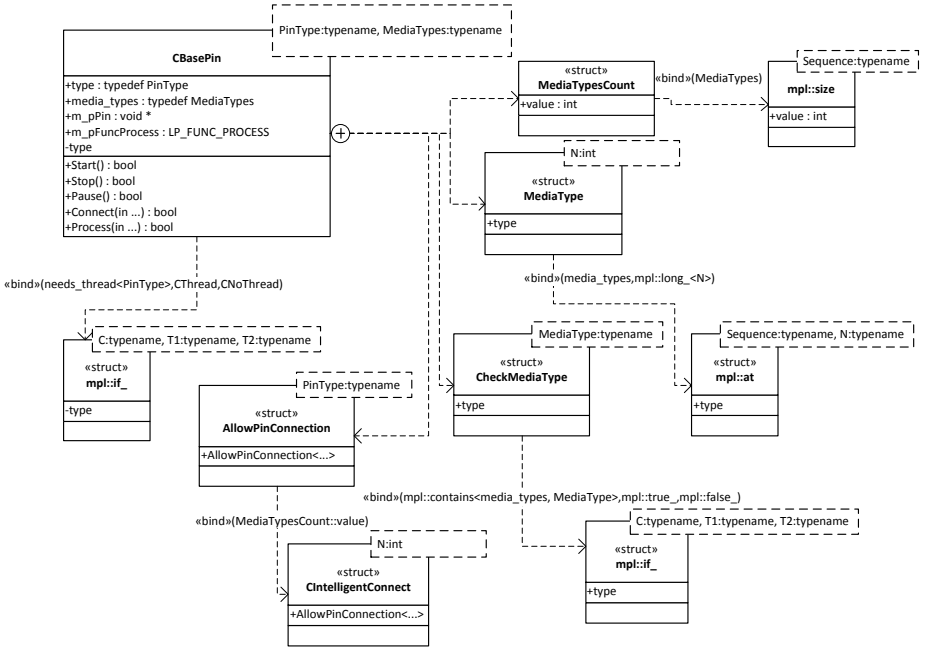


Figura 6.14: Diagrama de classes do *template* artefacto pino.

O *template* artefacto pino é representado pela classe *template* CBasePin. Todos os pinos que constituem os filtros da *framework* generativa de videovigilância herdam da classe *template* CBasePin. A classe *template* CBasePin recebe dois parâmetros de entrada de *template*, o tipo de dados PinType e o tipo de dados MediaTypes. O tipo de dados PinType define o tipo de pino a implementar, pino de entrada (input_pin_t) ou pino de saída (output_pin_t). O tipo de dados MediaTypes define um vetor com os tipos de *media* de áudio/vídeo suportados pelo pino.

A classe *template* CBasePin herda da classe CThread ou da classe CNoThread, dependendo do facto de o pino a implementar ser um pino de um filtro de entrada. Tal como foi referido anteriormente, os pinos dos filtros de entrada criam um novo fluxo de execução que são a fonte de dados para os restantes filtros que compõem

a cadeia de processamento do *filter graph*. O controlo da classe base é efetuado pela metafunção if_ da biblioteca *Boost MPL* tal como mostram as seguintes linhas de código:

```
template <typename PinType, typename MediaTypes>
class CBasePin
   : public mpl::if_<is_same<PinType::FilterType, source_filter_t>,
                     CThread, CNoThread>::type
{ ... };
```

O código seguinte apresenta a definição da classe CJPEGDecInputPin que implementa o pino de entrada do filtro de transformação CJPEGDecTransform que suporta o tipo de *media* de vídeo JPEG_MEDIA_TYPE:

```
class CJPEGDecInputPin
   : public CBasePin<input_pin_t,
               typename mpl::vector<JPEG_MEDIA_TYPE> >
{
public:
   CJPEGDecInputPin();
   ~CJPEGDecInputPin();
   ...
};
```

A classe CJPEGDecInputPin herda da classe *template* CBasePin. A classe *template* CBasePin recebe como parâmetros de entrada de *template* o tipo de pino (input_pin_t) e o vetor com os tipos de *media* suportados pelo pino.

***Interface* Estática do *Template* Artefacto Pino** Para implementar um novo pino, é necessário que o pino herde da classe *template* CBasePin, a qual define a seguinte *interface* estática:

i. O tipo de dados type, que indica qual o tipo de pino a implementar (pino de entrada ou pino de saída);

ii. O tipo de dados media_types, que define um vetor com o tipo de *media* de áudio/vídeo suportados pelo pino;

iii. Os métodos Start, Stop e Pause, que controlam o estado e o fluxo de execução do pino;

iv. O método Connect, que permite ligar o pino a um outro pino de diferentes filtros que pertencem ao mesmo *filter graph*;

iv. O método Process, que processa os dados enviados por um pino de um outro filtro e os envia a um outro pino de outro filtro caso este exista.

O código seguinte apresenta o *concept* PinConcept, que define a *interface* estática do *template* artefacto pino:

```
template <typename P>
struct PinConcept {
   void constraints() {
      typedef typename P::type type; // define pin type
      typedef typename P::media_types media_types; // define pin media types
      P.Start(); // P needs to implement Start method
      P.Stop(); // P needs to implement Stop method
      P.Pause(); // P needs to implement Pause method
      P.Connect(void*, LP_FUNC_PROCESS); // P needs to implement Connect method
      P.Process(void*, CMediaSample*); // P needs to implement Process method
   }
   P p;
};
```

Acesso aos Tipos de Media A classe *template* CBasePin implementa três metafunções que permitem obter informações sobre os tipos de *media* suportados por um determinado pino em tempo de compilação: MediaTypesCount, MediaType e CheckMediaType.

A metafunção MediaTypesCount devolve o número de tipos de *media* suportados pelo pino:

```
struct MediaTypesCount
   : public mpl::size<MediaTypes>
{ };
```

O número de tipos de *media* suportados pelo pino é obtido usando a metafunção size da biblioteca *Boost MPL* que devolve o número de elementos do vetor MediaTypes.

A metafunção MediaType devolve o tipo de *media* com índice N do pino:

```
template <int N>
struct MediaType
   : public mpl::at<media_types, mpl::long_<N> >
{ };
```

O tipo de *media* com índice N do pino é obtido através da metafunção at da biblioteca *Boost MPL*. A metafunção at recebe como parâmetros de entrada de *template* o vetor media_types com os tipos de *media* suportados pelo pino e o número inteiro N com o índice do tipo de *media* no vetor media_types.

A metafunção CheckMediaType verifica se o tipo de *media* MediaType é suportado pelo pino:

```
template <typename MediaType>
```

```
struct CheckMediaType
    : mpl::if_<mpl::contains<media_types, MediaType>, mpl::true_, mpl::false_>::type
{ };
```

Para verificar se o tipo de *media* MediaType é suportado pelo pino é utilizada a metafunção if_, onde a condição de teste é o resultado da metafunção contains da biblioteca *Boost MPL*. A metafunção contains devolve verdade se o tipo de *media* MediaType pertence ao vetor media_types que contém os tipos de *media* suportados pelo pino. Caso o resultado seja verdade, é devolvida a metafunção true_ da biblioteca *Boost MPL*, caso contrário é devolvida a metafunção false_ também da biblioteca *Boost MPL*.

As linhas de código seguintes verificam se a ligação entre o pino de saída CRTSPOutputPin do filtro de entrada CRTSPSource e o pino de entrada CJPEGDecInputPin do filtro de transformação CJPEGDecTransform do *filter graph* fg é possível. Para tal são comparados os tipos de *media* com índice 0 de cada um dos pinos usando a metafunção is_same da biblioteca *Boost*:

```
typedef fg.GetFilter<0>().GetPin<0>().MediaType<0>::type media_type1;
typedef fg.GetFilter<1>().GetPin<0>().MediaType<0>::type media_type2;
static_assert(!is_same<media_type1, media_type2>::value,
                                "The media types are not compatible!");
```

Ligação entre dois Pinos Para estabelecer a ligação entre dois pinos a classe *template* CBasePin define a metafunção AllowPinConnection e implementa o método Connect.

A metafunção AllowPinConnection verifica se a ligação entre dois pinos é possível:

```
template<typename PinType>
struct AllowPinConnection
    : CIntelligentConnect<MediaTypesCount::value>::template
          AllowPinConnection<CBasePin, PinType>
{ };
```

A metafunção recebe como parâmetro de entrada de *template* o tipo de dados PinType, que define o tipo de dados do pino ao qual se quer estabelecer a ligação. Para verificar se a ligação entre os dois pinos, CBasePin e PinType, é válida é utilizado o método AllowPinConnection da estrutura *template* CIntelligentConnect.

O método Connect estabelece a ligação entre dois pinos:

```
bool Connect(void *pPin, LP_FUNC_PROCESS pFuncProcess) {
    m_pPin = (void *)pPin;
    m_pFuncProcess = pFuncProcess;
```

```
    return true;
}
```

O método Connect recebe como parâmetros de entrada um apontador para o pino ao qual se vai estabelecer a ligação e um apontador para o método do pino que processa os dados de áudio/vídeo por forma a que este os possa processar, e eventualmente passar os dados que foram processados a um outro pino. O apontador para o pino e para a função de processamento do pino são gravados em duas variáveis da classe *template* CBasePin, m_pPin e m_pFuncProcess.

As linhas de código seguintes numa primeira fase verificam se a ligação entre os pinos do filtro com índice 0 (CRTSPSource) e do filtro com índice 1 (CJPEGDecTransform) do *filter graph* da Figura 6.11 é possível. Caso a ligação não seja possível, o compilador termina o processo de compilação e mostra uma mensagem de erro a informar que não é possível estabelecer a ligação entre os dois pinos. Caso a ligação seja possível, é estabelecida a ligação entre os dois pinos:

```
typedef fg.FilterType<0>::type filter_type_0;
typedef filter_type_0::PinType<0>::type pin_type_0;
typedef fg.FilterType<1>::type filter_type_1;
typedef filter_type_1::PinType<0>::type pin_type_1;

static_assert(!pin_type_0::AllowPinConnection<pin_type_1>::value,
      "It is not possible to connect the two pins.");

fg.GetFilter<0>().GetPin<0>().Connect(
      &fg.GetFilter<1>().GetPin<0>(), fg.GetFilter<1>().GetPin<0>()::Process);
```

Controlo do Fluxo de Execução do Pino As funções que controlam o fluxo de execução do pino são implementadas nos métodos da classe *template* CBasePin: Start, Stop e Pause.

O método Start inicia o fluxo de execução do pino:

```
bool Start() {
   return CThreadStrategy<is_same<PinType::FilterType,
      source_filter_t>::value >::template
         Start<CBasePin>(this);
}
```

Caso o filtro ao qual o pino pertence seja um filtro de entrada, o método Start da classe *template* CBasePin inicia uma nova *thread* que permite fazer a captura dos dados de forma independente da *thread* principal do sistema de videovigilância. Caso contrário, o método Start apenas devolve true. O processo de decisão de qual

das duas opções deve ser executada é gerido pela estrutura *template* CThreadStrategy no método *template* Start.

O método Stop termina o fluxo de execução do pino:

```
bool Stop() {
    return CThreadStrategy<is_same<PinType::FilterType,
        source_filter_t>::value >::template
            Stop<CBasePin>(this);
}
```

O método Pause faz uma pausa no fluxo de execução do pino:

```
bool Pause() {
    return CThreadStrategy<is_same<PinType::FilterType,
        source_filter_t>::value >::template
            Pause<CBasePin>(this);
}
```

Os métodos Stop e Pause não foram explicados, porque funcionam de forma análoga ao método Start.

Para iniciar o fluxo de execução dos pinos dos filtros que compõem o *filter graph* da Figura 6.11 são necessárias as seguintes linhas de código:

```
fg.GetFilter<0>().GetPin<0>().Start(); // CRTSPOutputPin
fg.GetFilter<1>().GetPin<0>().Start(); // CJPEGDecInputPin
fg.GetFilter<1>().GetPin<1>().Start(); // CJPEGDecOutputPin
fg.GetFilter<2>().GetPin<0>().Start(); // CXVInputPin
```

Processamento dos Dados de Áudio/Vídeo Para processar os dados enviados por um outro pino, a classe *template* CBasePin implementa o método estático Process:

```
static bool Process(void *pInstance, CMediaSample *pSample) {
    FilterType *pThis = (FilterType *)pInstance;
    return pThis->ProcessSample(pSample);
}
```

O método estático Process recebe como parâmetros de entrada o apontador para a instância do pino que recebe os dados de áudio/vídeo e um apontador para uma classe do tipo CMediaSample que contém os dados de áudio/vídeo a processar pelo pino. Como o método é estático, numa primeira fase é feito o *cast* para o tipo de dados do pino e, por fim, é executado o método ProcessSample da classe *template* CBasePin:

```
bool ProcessSample(CMediaSample *pSample) {
    // TODO: process the sample
```

```
    if (m_pFuncProcess) {
        // send the sample to connected pin
        m_pFuncProcess(m_pPin, pSample);
    }
    return true;
}
```

O método ProcessSample da classe *template* CBasePin recebe como parâmetro de entrada um apontador para o *media sample* a processar. Depois de o *media sample* ser processado pelo pino, é enviado ao pino que está ligado ao pino atual, caso este exista.

Template* Artefacto *Intelligent Connect O *Intelligent Connect* é a entidade responsável pelo processo de validação da ligação entre dois pinos e caso a ligação seja válida, estabelece a ligação entre eles. O *Intelligent Connect* suporta dois modos de ligação: o modo de ligação direta e o modo de ligação automática. No modo de ligação direta, se a ligação entre dois pinos não é válida, é apresentada uma mensagem de erro a informar que a ligação entre os dois pinos não é possível. No modo de ligação automática, mesmo que os tipos de *media* de áudio/vídeo dos dois pinos não sejam compatíveis, o *Intelligent Connect* faz o trabalho de procurar, adicionar e ligar os filtros de transformação intermédios ao *filter graph* de modo a que ligação entre os dois seja possível. Caso não seja encontrado um caminho que permita a ligação entre os dois pinos, é apresentada uma mensagem de erro a informar que a ligação entre os dois pinos não é possível.

O *Intelligent Connect* também é a entidade responsável pela gestão do processo de instanciação dos componentes necessários para obter uma configuração específica do sistema. Isto é, quando existem duas ou mais instanciações do mesmo filtro com diferentes configurações, são criadas duas ou mais instâncias do mesmo filtro produzindo código desnecessário (*code bloat*). O *Intelligent Connect* identifica este tipo de instanciações e minimiza o código desnecessário (*code bloat*) no ficheiro executável final da aplicação.

Diagrama de Classes A Figura 6.15 apresenta o diagrama de classes do *template* artefacto *Intelligent Connect*.

A estrutura CIntelligentConnect implementa o *template* artefacto *Intelligent Connect*. Para validar a ligação entre dois filtros, a estrutura CIntelligentConnect implementa a metafunção AllowFilterConnection. Para validar a ligação entre um filtro e um

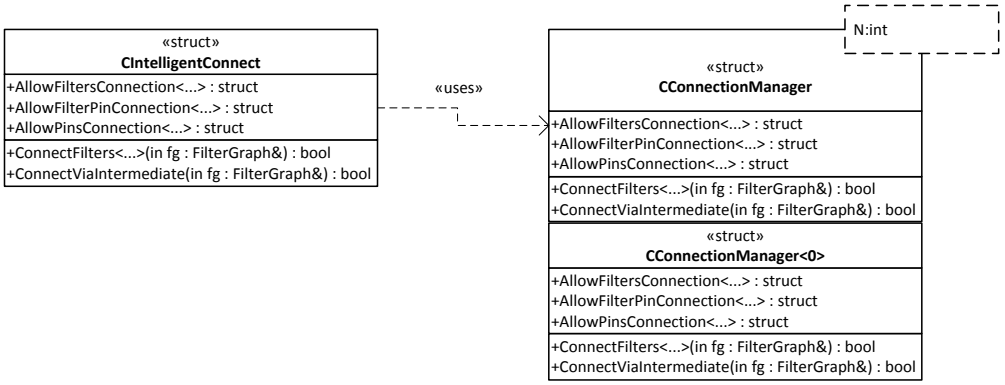


Figura 6.15: Diagrama de classes do *template* artefacto *Intelligent Connect*.

pino, a estrutura CIntelligentConnect implementa a metafunção AllowFilterPinConnection. Finalmente, para validar a ligação entre dois pinos, a estrutura CIntelligentConnect implementa a metafunção AllowPinConnection.

Para estabelecer as ligações entre os filtros que compõem um determinado *filter graph*, a estrutura CIntelligentConnect implementa dois métodos *template*: o método *template* ConnectFilters e o método *template* ConnectViaIntermediate. O primeiro utiliza o modo de ligação direta para ligar os filtros e o segundo utiliza o modo de ligação automática para ligar os filtros.

Todas as metafunções e métodos *template* da estrutura CIntelligentConnect utilizam as metafunções e métodos *template* correspondentes da estrutura *template* CConnectionManager. De seguida é feita uma breve descrição de cada uma da metafunções e métodos *template* da estrutura CIntelligentConnect.

Metafunção *AllowPinsConnection* A metafunção AllowPinsConnection verifica se a ligação entre dois pinos é válida. A ligação entre dois pinos só é válida se o pino que inicia o processo de ligação (pino ativo) for do tipo pino de saída, o pino que recebe a ligação (pino passivo) for do tipo pino de entrada e se existir algum tipo de *media* de áudio/vídeo compatível entre os dois pinos:

```
template<typename OutputPin, typename InputPin>
struct AllowPinsConnection {
   enum { value = (is_output_pin<OutputPin>::value & is_input_pin<InputPin>::value) ?
            CConnectionManager<OutputPin::MediaTypesCount::value>::template
                  AllowPinsConnection<OutputPin, InputPin>::value : 0 };
};
```

A metafunção recebe como parâmetros de entrada de *template* o tipo de dados OutputPin, que representa o tipo de dados do pino ativo, e o tipo de dados InputPin,

que representa tipo de dados do pino passivo. Como o pino ativo tem que ser do tipo pino de saída e o pino passivo tem que ser do tipo pino de entrada, é feito um teste inicial para verificar se o pino ativo e o pino passivo satisfazem a condição usando as metafunções is_output_pin e is_input_pin respetivamente. Se o resultado for falso, a metafunção retorna falso indicando que a ligação entre os dois pinos não é possível. Caso o resultado seja verdade, a metafunção retorna o resultado da metafunção AllowPinsConnection da estrutura *template* CConnectionManager.

Metafunção *AllowFilterPinConnection* A metafunção AllowFilterPinConnection verifica se a ligação entre um filtro (filtro ativo) e um pino (pino passivo) é válida. Para que a ligação seja válida, tem que existir na lista de pinos do filtro ativo um pino que seja compatível com o pino passivo:

```
template<typename OutputFilter, typename InputPin>
struct AllowFilterPinConnection
   : public CConnectionManager<OutputFilter::PinTypesCount::value>::template
                AllowFilterPinConnection<OutputFilter, InputPin>
{ };
```

A metafunção recebe como parâmetros de entrada de *template* o tipo de dados OutputFilter, que representa o tipo de dados do filtro ativo, e o tipo de dados InputPin, que representa o tipo de dados do pino passivo. Como a metafunção AllowFilterPinConnection herda da metafunção AllowFilterPinConnection da estrutura *template* CConnectionManager, o resultado final da metafunção AllowFilterPinConnection depende do resultado da metafunção AllowFilterPinConnection da estrutura *template* CConnectionManager.

Metafunção *AllowFiltersConnection* A metafunção AllowFiltersConnection verifica se a ligação entre dois filtros é válida. Para que a ligação seja válida, o filtro ativo tem que ser do tipo filtro de entrada ou do tipo filtro de transformação, o filtro passivo tem que ser do tipo filtro de transformação ou do tipo filtro de saída e tem que existir um pino na lista de pinos do filtro ativo que seja compatível com um pino na lista de pinos do filtro passivo:

```
template<typename OutputFilter, typename InputFilter>
struct AllowFiltersConnection
   : public CConnectionManager<OutputFilter::PinTypesCount::value>::template
        AllowFiltersConnection<OutputFilter, InputFilter>
{
   static_assert(is_valid_output_filter<OutputFilter>::value,
                        "It is not possible to use an invalid output filter!");
   static_assert(is_valid_input_filter<InputFilter>::value,
                        "It is not possible to use an invalid input filter!");
```

```
};
```

A metafunção recebe como parâmetros de entrada de *template* o tipo de dados OutputFilter, que representa o tipo de dados do filtro ativo, e o tipo de dados InputFilter, que representa o tipo de dados do filtro passivo.

Para validar os tipos de filtros são utilizadas as metafunções is_valid_output_filter e is_valid_input_filter. A metafunção is_valid_output_filter verifica se o filtro ativo é do tipo filtro de entrada ou do tipo filtro de transformação, ao passo que a metafunção is_valid_input_filter verifica se o filtro passivo é do tipo filtro de transformação ou do tipo filtro de saída. Se o resultado de uma das metafunções for falso, o processo de compilação termina e é apresentada uma mensagem de erro pelo compilador a informar que um dos filtros não é válido. Caso o resultado seja verdadeiro, o resultado final da metafunção AllowFiltersConnection depende de execução da metafunção AllowFiltersConnection da estrutura *template* CConnectionManager.

Método *ConnectFilters* O método *template* ConnectFilters estabelece as ligações entre os filtros que compõem um determinado *filter graph*. Ao utilizar o método ConnectFilters para estabelecer as ligações entre os filtros, todas as ligações são estabelecidas utilizando o modo de ligação direta. Isto é, se não for possível estabelecer a ligação entre dois filtros, é apresentada uma mensagem de erro pelo compilador e o processo de compilação termina. As linhas de código seguintes definem o método *template* ConnectFilters:

```
template<typename FilterTypes, typename FilterConnections, typename FilterGraph>
inline static bool ConnectFilters(FilterGraph& fg) {
   return CConnectionManager<mpl::size<FilterConnections>::value>::template
            ConnectFilters<FilterTypes, FilterConnections, FilterGraph>(fg);
}
```

O método *template* recebe como parâmetros de entrada de *template* o vetor FilterTypes com os tipos de dados dos filtros, o vetor FilterConnections com os pares ordenados das ligações entre os filtros e o tipo de dados FilterGraph que representa o tipo de dados do *filter graph*. Como parâmetro de entrada do método, recebe uma referência para o *filter graph* para aceder à informação de cada um dos filtros. As ligações entre os filtros são estabelecidas no método ConnectFilters da estrutura *template* CConnectionManager, que utiliza a recursividade para estabelecer as ligações definidas nos pares ordenados que constituem o vetor FilterConnections.

Método *ConnectViaIntermediate* O método *template* ConnectViaIntermediate estabelece as ligações entre os filtros que compõem um determinado *filter graph* utilizando o método de ligação automática. Isto é, se alguma ligação entre dois filtros não for possível devido à incompatibilidade de pinos, o *Intelligent Connect* faz o trabalho de procurar, adicionar e ligar os filtros de transformação intermédios ao *filter graph* de modo a que ligação entre os dois seja possível. As linhas de código seguintes definem o método *template* ConnectViaIntermediate:

```
template<typename FilterTypes, typename FilterConnections, typename FilterGraph>
inline static bool ConnectViaIntermediate(FilterGraph& fg) {
   return CConnectionManager<mpl::size<FilterConnections>::value>::template
            ConnectViaIntermediate<FilterTypes, FilterConnections, FilterGraph>(fg);
}
```

O método *template* ConnectViaIntermediate funciona de forma análoga ao método *template* ConnectFilters mas, em vez de utilizar a metafunção ConnectFilters da estrutura *template* CConnectionManager, utiliza a metafunção ConnectViaIntermediate da estrutura *template* CConnectionManager.

Metafunção *InstantiationManager* Quando a aplicação de videovigilância é composta por mais do que um *filter graph*, em que em cada um dos *filter graphs* contém o mesmo filtro mas com diferentes configurações, no ato de compilação da aplicação é instanciada uma classe diferente para cada um deles, o que se traduz em código desnecessário (*code bloat*). A metafunção InstantiationManager da estrutura CIntelligentConnect permite eleminar o código desnecessário (*code bloat*) produzido por configurações deste tipo.

Para uma melhor compreensão, é dado como exemplo a configuração da aplicação da Figura 6.16 constituída por dois (*filter graphs*): o *filter graph* 1 e o *filter graph* 2. O *filter graph* 1 faz a captura de uma câmara IP que utiliza o protocolo RTSP para enviar imagens de vídeo no formato H.264 com o *profile base*. Por sua vez, o *filter graph* 2 faz a captura de uma câmara IP que usa o protocolo HTTP para enviar imagens de vídeo no formato H.264 com o *profile main*. Como cada uma das classes *template* CH264DecTransform tem diferentes parâmetros de entrada de *template*, é criada uma instância diferente da classe para cada um dos casos. Como a maior parte do código da classe é comum às duas configurações, o código da aplicação resultante tem o dobro do tamanho realmente necessário (code desnecessário). Para resolver este problema, a metafunção InstantiationManager numa primeira fase analisa o tipo de filtros e as suas configurações em cada um dos *filter graphs*. Se existirem filtros com o mesmo tipo mas com diferentes configurações em vários

filter graphs, é criada uma lista de configurações para o filtro em questão de forma a que seja criada uma só instância da classe.

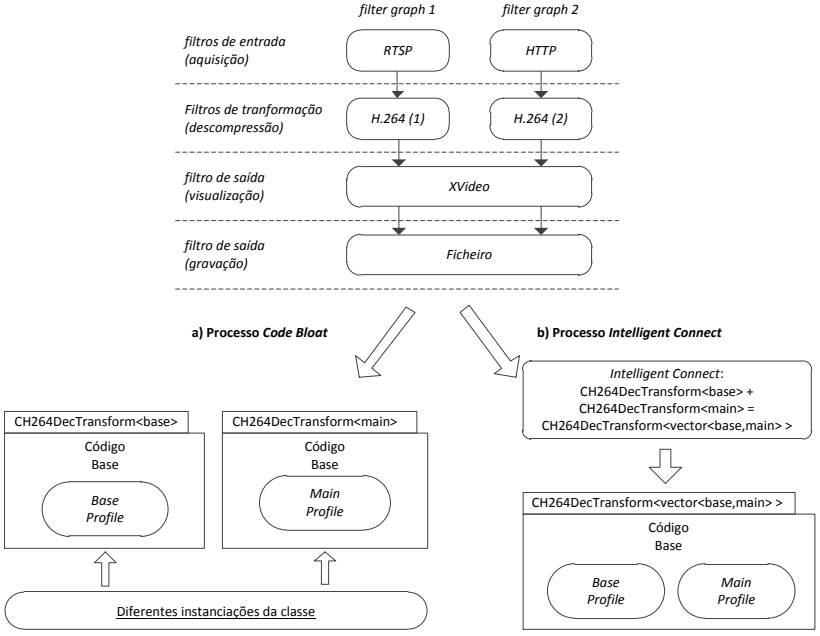


Figura 6.16: Processo Código *Bloat* e *Intelligent Connect*.

Connection Manager O *Connection Manager* é utilizado pelo *Intelligent Connect* para validar e estabelecer as ligações dos filtros que compõem um determinado *filter graph* de forma iterativa usando a recursividade. Para isso, a estrutura *template* CConnectionManager implementa uma *template* genérica que inicia o processo de validação e execução das ligações e uma *template* especializada que termina o processo de validação e execução das mesmas. As metafunções e os métodos implementados pela estrutura *template* CConnectionManager seguem o mesmo padrão de implementação e execução da estrutura *template* CFilterGraphManager.

Metafunção *AllowPinsConnection* Para iniciar o processo de validação da ligação entre dois pinos é definida na *template* genérica da estrutura *template* CConnectionManager o método *template* AllowPinsConnection:

```
template<int N>
struct CConnectionManager {
    ...
    template<typename OutputPin, typename InputPin>
    struct AllowPinsConnection {
        typedef typename OutputPin::MediaType<N−1>::type media_type;
        enum { value = InputPin::CheckMediaType<media_type>::value ? 1
                    : CConnectionManager<N−1>::template
```

```
                AllowPinsConnection<OutputPin, InputPin>::value };
    };
    ...
};
```

A estrutura *template* CConnectionManager recebe como parâmetro de entrada o número de tipos de *media* de áudio/vídeo suportados pelo pino ativo OutputPin. Assim, em cada iteração é testado se o tipo de *media* de áudio/vídeo com índice N-1 é suportado pelo pino passivo InputPin através do método *template* CheckMediaType. Se o resultado do método *template* CheckMediaType for verdade, a metafunção AllowPinsConnection devolve verdade, indicando que a ligação entre os dois pinos é possível. Caso contrário, o resultado depende da execução da mesma metafunção de forma recursiva com o índice N-1, de forma a testar todos os tipos de *media* de áudio/vídeo do pino ativo. O código seguinte implementa a condição de paragem:

```
template<>
struct CConnectionManager<0> {
    ...
    template<typename OutputPin, typename InputPin>
    struct AllowPinsConnection {
        enum { value = 0 };
    };
    ...
};
```

A condição de paragem é executada quando todos os tipos de *media* de áudio/vídeo suportados pelo pino ativo foram testados e nenhum é compatível com os tipos de *media* de áudio/vídeo suportados pelo pino passivo. Neste caso, o resultado da metafunção é falso, indicando que a ligação entre os pinos não é possível.

As linhas de código seguintes verificam se a ligação entre o pino de saída CRTSPOutputPin do filtro de entrada CRTSPSource e o pino de entrada CJPEGDecInputPin do filtro de transformação CJPEGDecTransform é possível:

```
static_assert(CIntelligentConnect::AllowPinsConnection<CRTSPOutputPin,
            CJPEGDecInputPin>::value, "It is not possible connect the two pins!");
```

A Figura 6.17 apresenta o processo iterativo do exemplo apresentado:

Como ambos os pinos suportam o tipo de *media* de vídeo JPEG_MEDIA_TYPE, a metafunção AllowPinsConnection retorna verdade.

Metafunção *AllowFilterPinConnection* Para iniciar o processo de validação da ligação entre um filtro (filtro ativo) e um pino (pino passivo), é definida

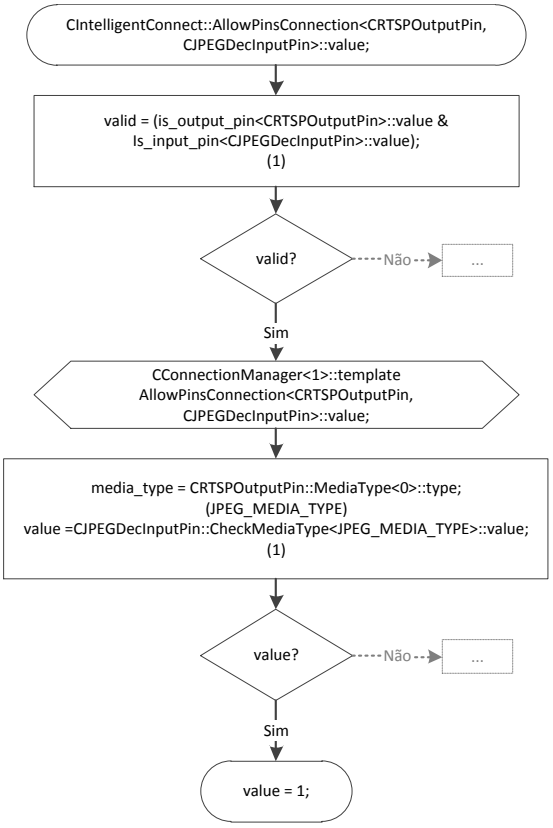


Figura 6.17: Processo iterativo de verificação da ligação entre os pinos CRTSPOutputPin e CJPEGDecInputPin.

na *template* genérica da estrutura *template* CConnectionManager o método *template* AllowFilterPinConnection:

```
template<int N>
struct CConnectionManager {
    ...
    template<typename OutputFilter, typename InputPin>
    struct AllowFilterPinConnection {
    private:
        typedef typename OutputFilter::PinType<N−1>::type pin_type;
        enum { found = InputPin::AllowPinConnection<pin_type>::value };
    public:
        enum { value = found ? 1 : CConnectionManager<N−1>::template
                    AllowFilterPinConnection<OutputFilter, InputPin>::value,
               input = found ? N−1 : CConnectionManager<N−1>::template
                    AllowFilterPinConnection<OutputFilter, InputPin>::input };
    };
    ...
};
```

A estrutura *template* CConnectionManager recebe como parâmetro de entrada o número de pinos do filtro ativo OutputFilter. Desta forma, em cada iteração é testado se o pino com índice N-1 do filtro ativo é compatível com o pino passivo InputPin

usando a metafunção do pino AllowPinConnection. Se o resultado da metafunção AllowPinConnection for verdade, a metafunção AllowPinsConnection da estrutura *template* CConnectionManager retorna verdade, indicando que a ligação entre o pino com índice N-1 do filtro ativo e o pino passivo é possível (variável value). Quando a ligação é possível, a variável input indica o índice do pino do filtro ativo que é compatível com o pino passivo. Se a ligação não for possível, o resultado da metafunção AllowPinsConnection da estrutura *template* CConnectionManager depende da execução da mesma metafunção de forma recursiva com o índice N-1, possibilitando o teste de compatibilidade de todos os pinos do filtro ativo com o pino passivo. O código seguinte implementa a condição de paragem:

```
template<>
struct CConnectionManager<0> {
    ...
    template<typename OutputFilter, typename InputPin>
    struct AllowFilterPinConnection {
        enum { value = 0,
               input = -1 };
    };
    ...
};
```

A condição de paragem é executada quando todos os pinos do filtro ativo foram testados e nenhum foi compatível com o pino passivo. Neste caso, o resultado da metafunção é falso, indicando que a ligação entre o filtro ativo OutputFilter e o pino passivo InputPin não é possível.

Para verificar se é possível a ligação entre o filtro de transformação CJPEGDecTransform e o pino de entrada CXVInputPin do filtro de saída CXVRenderer, são apresentadas as seguintes linhas de código:

```
static_assert(AllowFilterPinConnection<CJPEGDecTransform, CXVInputPin>::value,
        "It is not possible to connect the given filter and the given pin!");
```

A Figura 6.18 apresenta o processo iterativo da metafunção AllowFilterPinConnection, que verifica se a ligação entre o filtro CJPEGDecTransform e o pino CXVInputPin é possível:

Como o filtro CJPEGDecTransform contém o pino CJPEGDecOutputPin que suporta o mesmo tipo de *media* de vídeo JPEG_MEDIA_TYPE que o pino CXVInputPin, a metafunção AllowFilterPinConnection retorna verdade.

Metafunção *AllowFiltersConnection* Para iniciar o processo de validação da ligação entre dois filtros, é definido na *template* genérica da estrutura *template*

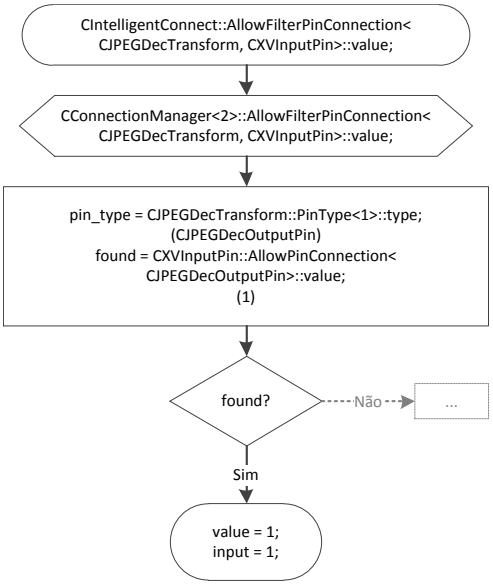


Figura 6.18: Processo iterativo de verificação da ligação entre o filtro CJPEGDecTransform e o pino CXVInputPin.

CConnectionManager o método *template* AllowFiltersConnection:

```
template<int N>
struct CConnectionManager {
    ...
    template<typename OutputFilter, typename InputFilter>
    struct AllowFiltersConnection {
    private:
        typedef typename OutputFilter::PinType<N−1>::type pin_type;
        enum { found = InputFilter::AllowPinConnection<pin_type>::value };
    public:
        enum { value = found ? 1 : CConnectionManager<N−1>::template
                    AllowFiltersConnection<OutputFilter, InputFilter>::value,
               output = found ? N−1 : CConnectionManager<N−1>::template
                    AllowFiltersConnection<OutputFilter, InputFilter>::output,
               input = found ? InputFilter::AllowPinConnection<pin_type>::input
                        : CConnectionManager<N−1>::template
                            AllowFiltersConnection<OutputFilter, InputFilter>::input };
    };
    ...
};
```

A estrutura *template* CConnectionManager recebe como parâmetro de entrada o número de pinos do filtro ativo OutputFilter. Assim, em cada iteração é testado se é possível ligar o pino com índice N-1 do filtro ativo com um dos pinos do filtro passivo InputFilter usando a metafunção do filtro AllowPinConnection. Se o resultado da metafunção AllowPinConnection for verdade, a metafunção AllowFiltersConnection da estrutura *template* CConnectionManager retorna verdade, indicando que a ligação entre o pino com índice N-1 do filtro ativo e um dos pinos do filtro passivo é possível

(variável value). Quando a ligação entre os filtros é possível, a variável output indica o índice do pino do filtro ativo e a variável input indica o índice do pino do filtro passivo. Se a ligação não for possível, o resultado da metafunção é o resultado da chamada da mesma metafunção de forma recursiva com o índice N-1 de forma a testar todos os pinos do filtro ativo com os pinos do filtro passivo. O código seguinte implementa a condição de paragem:

```
template<>
struct CConnectionManager<0> {
    ...
    template<typename OutputFilter, typename InputFilter>
    struct AllowFiltersConnection {
        enum { value = 0,
               output = −1,
               input = −1 };
    };
    ...
};
```

A condição de paragem é executada quando não existe um pino do filtro ativo que seja compatível com um pino do filtro passivo. Neste caso, o resultado da metafunção AllowFiltersConnection da estrutura *template* CConnectionManager é falso, indicando que a ligação entre os dois filtros não é possível.

As linhas de código seguintes verificam se a ligação entre o filtro de entrada CRTSP-Source e o filtro de saída CXVRenderer é possível:

```
static_assert(CIntelligentConnect::AllowFiltersConnection<CRTSPSource,
        CXVRenderer>::value, "It is not possible to connect the two filters!");
```

A Figura 6.19 apresenta o processo iterativo da metafunção AllowFiltersConnection, que verifica se a ligação entre o filtro CRTSPSource e o filtro CXVRenderer é possível:

A metafunção AllowFiltersConnection retorna falso, porque não existe nenhum pino que seja compatível entre o filtro de entrada CRTSPSource e o filtro de saída CXVRenderer.

Método *ConnectFilters* O método *template* ConnectFilters estabelece as ligações entre os filtros que compõem um *filter graph*. Ao estabelecer as ligações entre os filtros, é utilizado o método direto de ligações, isto é, se alguma das ligações não for válida, o processo de compilação termina e é apresentada uma mensagem de erro a explicar o sucedido. As linhas de código seguintes apresentam o método *template* ConnectFilters da *template* genérica:

```
template<int N>
struct CConnectionManager {
```

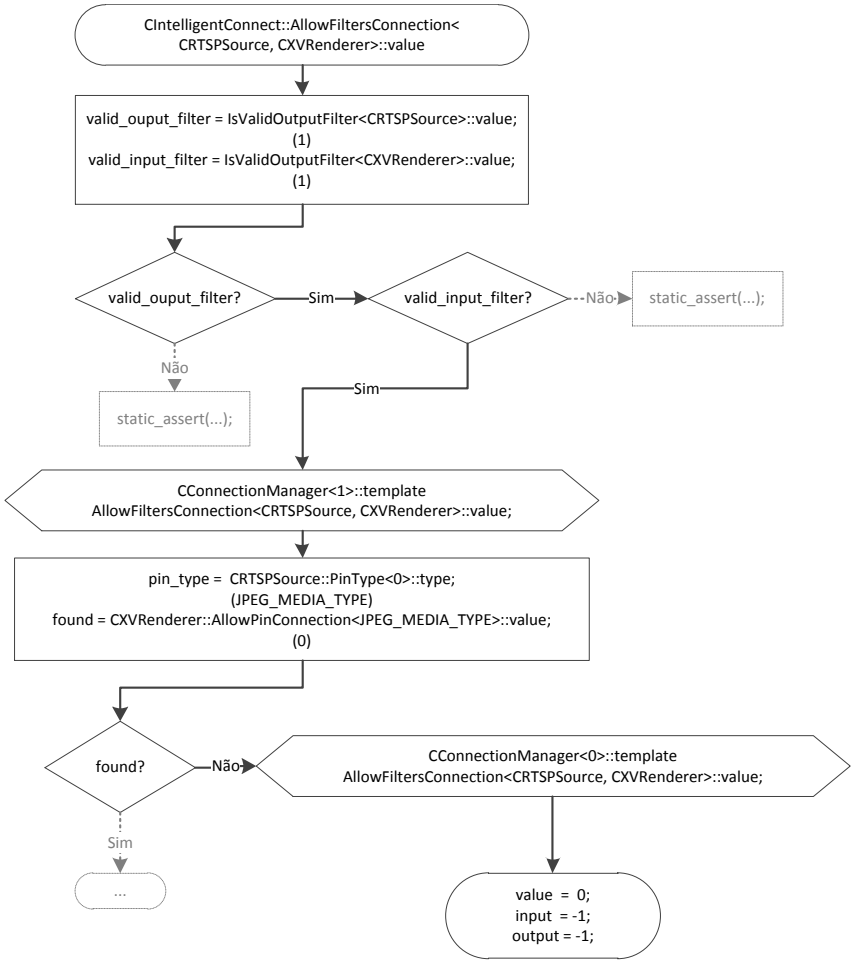


Figura 6.19: Processo iterativo de verificação da ligação entre o filtro CRTSPSource e o filtro CXVRenderer.

```
    ...
    template<typename FilterTypes, typename FilterConnections, typename FilterGraph>
    inline static bool ConnectFilters(FilterGraph& fg) {
        typedef typename mpl::at<FilterConnections, mpl::long_<N-1> >::type Connection;
        typedef typename mpl::at<FilterTypes,
                                 mpl::first<Connection>::type >::type OutputFilter;
        typedef typename mpl::at<FilterTypes,
                                 mpl::second<Connection>::type >::type InputFilter;

        return (CConnectionManager<N-1>::template
                ConnectFilters<FilterTypes, FilterConnections, FilterGraph>(fg) &
                   fg.GetFilter<mpl::first<Connection>::type::value>().Connect<InputFilter>(
                      fg.GetFilter<mpl::second<Connection>::type::value>()));
    }
    ...
};
```

O método *template* recebe como parâmetros de entrada de *template* o vetor FilterTypes com os tipos de dados dos filtros, o vetor FilterConnections com os pares

ordenados das ligações entre os filtros e o tipo de dados FilterGraph que representa o tipo de dados do *filter graph*. Como parâmetro de entrada do método, recebe uma referência para o *filter graph* para aceder à informação de cada um dos filtros.

A variável Connection representa o par ordenado de índice N-1 do vetor de ligações FilterConnections. A variável OutputFilter representa o primeiro elemento do par ordenado de índice N-1, que indica o índice do filtro que estabelece a ligação no vetor FilterTypes. Finalmente, a variável InputFilter representa o segundo elemento do par ordenado de índice N-1, que indica o índice do filtro que recebe a ligação no vetor FilterTypes.

Para estabelecer a ligação entre o filtro com o tipo de dados OutputFilter e o filtro com o tipo de dados InputFilter é usado o método Connect do filtro que estabelece a ligação, passando como parâmetro de entrada o filtro que recebe a ligação. Para aceder à referência de cada um dos filtros é usado o método GetFilter da classe *template* CFilterGraph. A condição de paragem do método *template* ConnectFilters é definido na *template* especializada da estrutura *template* CConnectionManager:

```
template<>
struct CConnectionManager<0> {
    ...
    template<typename FilterTypes, typename FilterConnections, typename FilterGraph>
    inline static bool ConnectFilters(FilterGraph& fg) {
        return true;
    }
    ...
};
```

A Figura 6.20 apresenta o processo iterativo das ligações do *filter graph* da Figura 6.8.

Quando o método *template* ConnectFilters da estrutura *template* CIntelligentConnect é instanciado, são estabelecidas as ligações entre os filtros: índice 0 (CRTSPSource) e índice 1 (CJPEGDecTRansform) do *filter graph* e índice 1 (CJPEGDecTRansform) e índice 2 (CXVrenderer) do *filter graph*.

O resultado final do método *template* ConnectFilters é apresentado nas linhas de código seguintes em pseudo-código:

```
struct CIntelligentConnect_XYZ {
    inline static bool ConnectFilters(FilterGraph& fg) {
                // CRTSPSource -> CJPEGDecTransform
        return fg.GetFilter<0>().Connect(fg.GetFilter<1>()) &
                    // CJPEGDecTransform -> CXVRenderer
                    fg.GetFilter<1>().Connect(fg.GetFilter<2>());
    }
};
```

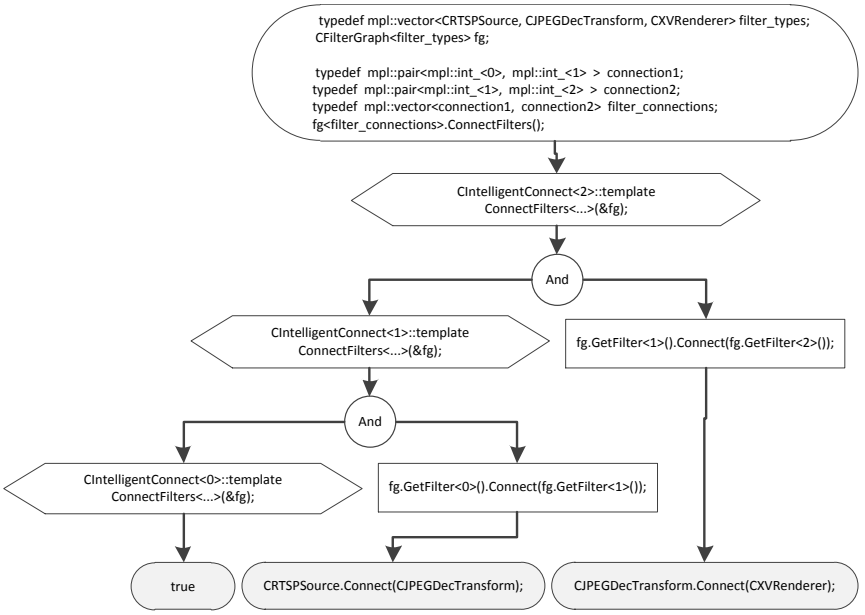


Figura 6.20: Processo iterativo das ligações do *filter graph* da Figura 6.8.

Método *ConnectViaIntermediate* O método *template* ConnectViaIntermediate estabelece as ligações entre os filtros que compõem um *filter graph* mas, em vez de utilizar o modo de ligação direta, utiliza o modo de ligação automática. Isto é, se a ligação entre dois filtros do *filter graph* não for possível utilizando o modo de ligação direta, o *connection manager* procura e adiciona os filtros de transformação intermédios que tornam a ligação entre os dois filtros possível. As linhas de código seguintes apresentam o método *template* ConnectViaIntermediate da *template* genérica:

```
template<int N>
struct CConnectionManager {
    ...
    template<typename FilterTypes, typename FilterConnections, typename FilterGraph>
    inline static bool ConnectViaIntermediate(FilterGraph& fg) {
        typedef typename mpl::at<FilterConnections, mpl::long_<N-1> >::type Connection;
        typedef typename mpl::at<FilterTypes, mpl::first<Connection>::type >::type OutputFilter;
        typedef typename mpl::at<FilterTypes, mpl::second<Connection>::type >::type InputFilter;

        typedef typename mpl::if_c<AllowFiltersConnection<OutputFilter, InputFilter>::value,
                IdentityFilters<FilterTypes, FilterConnections, FilterGraph>::type,
                AddIntermediateFilters<OutputFilter, InputFilter,
                        FilterTypes, FilterConnections, FilterGraph>::type>::type ConnectionInfo;

        return (CConnectionManager<N-1>::template
                ConnectViaIntermediate<ConnectionInfo::FilterTypes, ConnectionInfo::FilterConnections,
                        ConnectionInfo::FilterGraph>(fg) &
                                fg.GetFilter<mpl::first<ConnectionInfo::FilterConnections>
                                  ::type::value>().Connect<InputFilter>(
                                    fg.GetFilter<mpl::second<ConnectionInfo::FilterConnections>::type::value>()));
    }
    ...
```

```
};
```

O método *template* recebe como parâmetros de entrada de *template* o vetor FilterTypes com os tipos de dados dos filtros, o vetor FilterConnections com os pares ordenados das ligações entre os filtros e o tipo de dados FilterGraph que representa o tipo de dados do *filter graph*. Como parâmetro de entrada do método, recebe uma referência para o *filter graph* para aceder à informação de cada um dos filtros.

A variável Connection representa o par ordenado de índice N-1 do vetor de ligações FilterConnections. A variável OutputFilter representa o primeiro elemento do par ordenado de índice N-1, que indica o índice do filtro que estabelece a ligação no vetor FilterTypes. Finalmente, a variável InputFilter representa o segundo elemento do par ordenado de índice N-1, que indica o índice o índice do filtro que recebe a ligação no vetor FilterTypes.

Para verificar se é possível a ligação entre o filtro OutputFilter e o filtro InputFilter, é executada a metafunção AllowFiltersConnection. Se a ligação entre os dois filtros for possível, a metafunção if_c da biblioteca *Boost MPL* devolve a identidade dos tipos de filtros, das ligações e do tipo de dados do *filter graph*. Caso contrário, são adicionados ao *filter graph* os filtros intermédios que possibilitam a ligação entre os dois filtros, caso seja possível, através da metafunção AddIntermediateFilters.

Para estabelecer a ligação entre os filtros é usado o método Connect do filtro que estabelece a ligação, passando como parâmetro de entrada o filtro que recebe a ligação. Para aceder à referência de cada um dos filtros é usado o método GetFilter da classe *template* CFilterGraph. A condição de paragem do método *template* ConnectFilters é definido na *template* especializada da estrutura *template* CConnectionManager:

```
template<>
struct CConnectionManager<0> {
    ...
    template<typename FilterTypes, typename FilterConnections, typename FilterGraph>
    inline static bool ConnectViaIntermediate(FilterGraph& fg) {
        return true;
    }
    ...
};
```

6.2.2 Engenharia da Aplicação

Enquanto o domínio de engenharia se concentra no desenvolvimento de componentes reutilizáveis ("*development for reuse*"), a engenharia da aplicação concentra-se

na seleção e reutilização dos componentes ("*development with reuse*") a partir de um repositório de componentes para obter uma configuração específica de um sistema de videovigilância.

A engenharia da aplicação é dividida em duas atividades principais: (1) estabelecer um canal de comunicação entre os *stakeholders* (os clientes do sistema de videovigilância) e os analistas e programadores do sistema de forma a obter uma instância do modelo da família de produtos da videovigilância, e (2) gerar de forma automática o código executável final da aplicação a partir do modelo da família de produtos configurado anteriormente. A Figura 6.21 apresenta um diagrama onde é possível identificar as duas atividades principais da engenharia da aplicação.

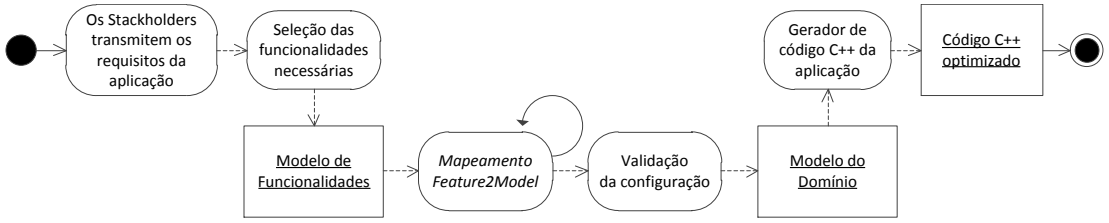


Figura 6.21: Diagrama com as duas atividades principais da engenharia da aplicação.

No final do processo de configuração do sistema de videovigilância e antes do processo de geração automática de código, é feito o mapeamento automático entre as funcionalidades do modelo de funcionalidades e os elementos do modelo do domínio.

Modelo de Funcionalidades

O modelo de funcionalidades é composto por um conjunto de funcionalidades que permitem obter uma configuração especifica da aplicação de videovigilância. Para tal, os *stakeholders* devem transmitir aos analistas e programadores do sistema os requisitos necessários da aplicação, para que estes possam selecionar as funcionalidades da *framework* generativa necessárias para obter a configuração pretendida.

A Figura 6.22 apresenta um exemplo de uma configuração transmitida por um *stakeholder* a um programador do sistema:

A configuração é constituída apenas por um *filter graph* que é composto por três filtros: o filtro de entrada CV4L2Source, o filtro de transformação CH264EncTransform e o filtro de saída CFileRenderer. O filtro de entrada CV4L2Source faz a captura de



Figura 6.22: Exemplo de uma configuração do *filter graph* transmitida por um *stakeholder*.

imagens descomprimidas de câmaras analógica no sistema operativo Linux. O filtro de transformação CH264EncTransform comprime as imagens no formato H.264. Finalmente, o filtro de saída CFileRenderer grava as imagens comprimidas no formato H.264 num ficheiro.

Depois de o *stakeholder* transmitir ao programador do sistema os requisitos da aplicação, o programador do sistema seleciona os componentes necessários a partir de um repositório de componentes para obter a configuração pretendida. Ao mesmo tempo, são definidas as propriedades de cada um dos componentes. A Figura 6.23 apresenta os componentes selecionados e as suas configurações no modo de modelação.

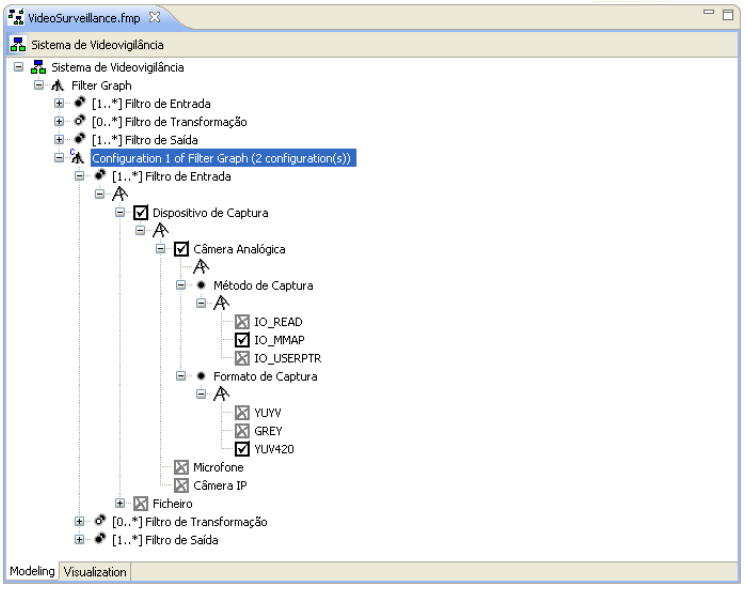


Figura 6.23: Seleção e configuração dos componentes no modo de modelação.

O exemplo apresenta a configuração com o nome *configuration 1 of Filter Graph* onde só é possível visualizar a seleção do filtro de entrada V4L2, visto que os outros filtros não são visíveis na figura. O filtro de entrada V4L2 está configurado com o método de captura IO_MMAP que utiliza a memória partilhada para captura de imagens e o formato de captura de imagem YUV420.

Depois de selecionados e configurados os componentes no modo de modelação é necessário estabelecer as ligações entre os componentes (filtros). Para isso é necessário mudar do modo de modelação para o modo de visualização. A Figura 6.24 apresenta as ligações entre os componentes selecionados no modo de visualização.

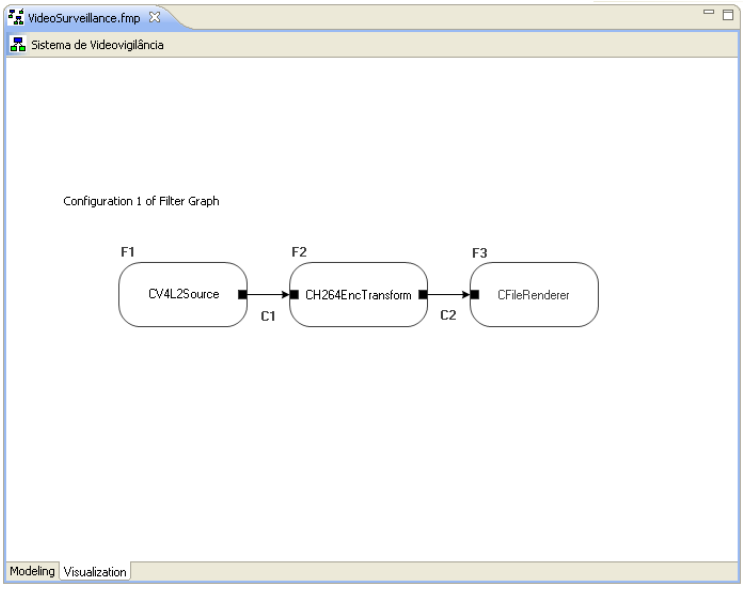


Figura 6.24: Ligação entre os componentes no modo de visualização.

Para que os dados da configuração possam ser utilizados pelo gerador de código da aplicação, é utilizada a notação XML, sendo os dados XML gravados no ficheiro cinfig.xml. Como os dados no formato XML podem ser editados manualmente num editor XML, é utilizado um validador config.xsd que valida o conteúdo do ficheiro cinfig.xml. Caso os dados não estejam corretos, é apresentada uma mensagem de erro e o processo de geração de código automático da aplicação termina.

As linhas de código seguintes apresentam um excerto do conteúdo do ficheiro XML da configuração da Figura 6.23:

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
  <FilterGraph>
    <Filters>
      <Filter name="CV4L2Source">
        <Method>mmap</Method>
        <Format>YUV420</Format>
      </Filter>
      <Filter name="CH264EncTransform"/>
      <Filter name="CFileRenderer"/>
    </Filters>
    <Connections>
      <Connection src="0" dst="1"/>
      <Connection src="1" dst="2"/>
```

```
    </Connections>
  </FilterGraph>
</Configuration>
```

Se no processo de seleção de componentes o programador do sistema identificar que uma das funcionalidades necessárias para obter a configuração pretendida pelo *stakeholder* aquando da transmissão dos requisitos da aplicação não é suportada pela *framework*, devem-se utilizar técnicas *Agile* e técnicas SPL para adicionar a funcionalidade, tal como ilustra a Figura 6.25.

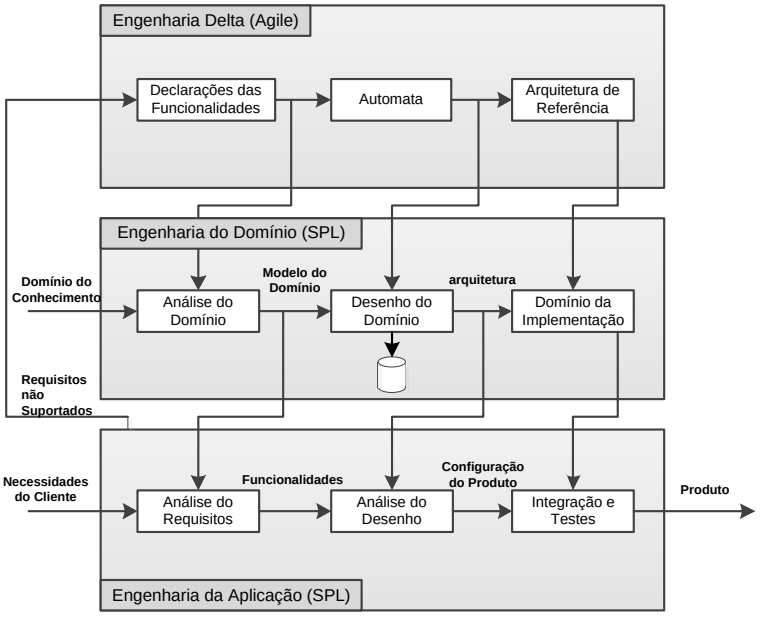


Figura 6.25: Integração de *Agile* e SPL no processo de desenvolvimento.

A utilização de técnicas de desenvolvimento *Agile* permite que novas funcionalidades sejam desenvolvidas e adicionadas à *framework* generativa de videovigilância de forma progressiva. Assim, em vez de se executar cada uma das fases de desenvolvimento em modo sequencial (por exemplo, a fase do desenho só vem depois de a fase de análise estar terminada) executa-se cada uma das fases de forma iterativa e incremental. Desta forma, é possível um contacto mais próximo com a necessidade do cliente e testar a funcionalidade inserida na *framework* de modo progressivo sem comprometer a fiabilidade do sistema.

Mapeamento entre o Modelo de Funcionalidades e o Modelo do Domínio

A abordagem MDD apresentada combina o modelo de funcionalidade e o modelo do domínio ao nível de *template* artefactos codificados em C++ *template meta-*

programming. O mapeamento entre o modelo de funcionalidades e o modelo do domínio é realizado de forma automática em três níveis de granularidade, isto é, ao nível do *template* artefacto *filter graph*, ao nível do *template* artefacto filtro e internamente no *template* artefacto filtro em diferentes pontos de variabilidade, tais como atributos e métodos quando o modo de sincronização é selecionado.

As funcionalidades selecionadas de um filtro vão sendo coletadas de forma sucessiva até ao momento em que o filtro possa ser representado por uma meta-expressão que representa um *template* artefacto em particular, sendo assim efetuado o processo de mapeamento. Outras funcionalidades que sejam selecionadas são mapeadas em pontos de variabilidades internos ou atributos no *template* artefacto filtro previamente mapeado por meta-expressões ou condições de presença, respetivamente, indicando onde estas devem ser injetadas ou removidas de uma instância especializada de *template*. O mapeamento ao nível do *template* artefacto *filter graph* é acionado por meta-expressões quando pelo menos dois filtros são ligados na *stream*.

Funcionalidades que não sejam suportadas ou cujo mapeamento não é possível no modelo do domínio, forçam a que o modelo do domínio seja estendido e analisado com cuidado. De momento, um *online partial evaluator* baseado em C++ *template metaprogramming* resolve com sucesso a integração das funcionalidades não suportadas pela *framework*, depois de feito o *refactoring* do código *legacy* de alguns filtros implementados em bibliotecas externas (Anisko, 2002). A Figura 6.26 mostra um exemplo simples da atividade *Feature2ModelMapping*.

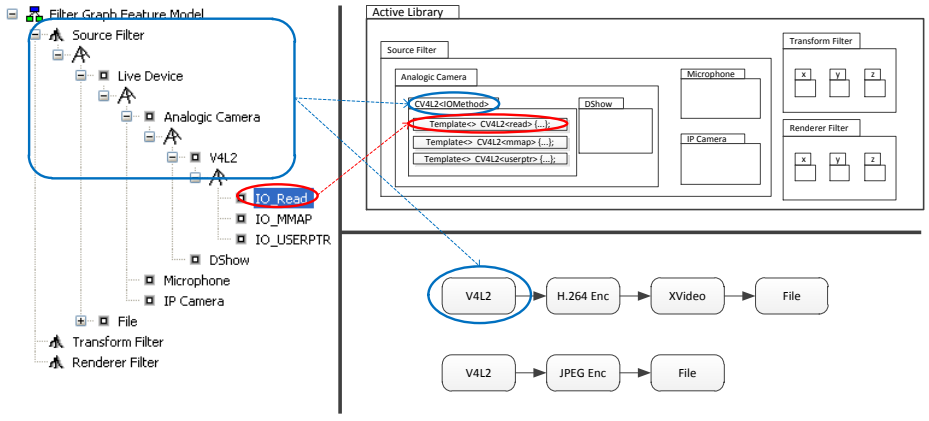


Figura 6.26: Exemplo simples da atividade *Feature2ModelMapping*.

Geração Automática do Código Final da Aplicação

De modo a gerar o código da aplicação de videovigilância é utilizado numa primeira fase um transformador XSLT (*eXtensible Stylesheet Language for Transformation*). O transformador XSLT recebe à entrada um ficheiro de configurações que contém as configurações do sistema no formato XML e as transforma em meta-expressões que são gravadas à saída do transformador no ficheiro de configurações Config.h.

As linhas de código seguintes apresentam o conteúdo do ficheiro de configurações Config.h gerado pelo transformador XSLT ao processar o ficheiro XML com as configurações do sistema da Figura 6.23.

```
struct Config {
   typedef mpl::vector<
      CV4L2Source<v4l2_mmap_method_t, v4l2_yuv420_format_t>,
      CH264EncTransform<...>,
      CFileRenderer<...> > fg1;

   typedef mpl::vector<
      mpl::pair<0,1>,
      mpl::pair<1,2>,
      mpl::pair<2,3> > c1;

   typedef mpl::vector<
      mpl::pair<fg1, c1> > fgs;
};
```

O ficheiro de configurações Config.h define apenas uma estrutura com o nome Config. Para cada *filter graph* a estrutura define um vetor com o tipo de dados dos filtros e um vetor com as ligações entre eles. Neste caso, como a aplicação contém apenas um *filter graph*, é definido o tipo de dados fg1, que representa um vetor com os tipos de dados dos filtros do *filter graph*, e o tipo de dados c1, que define um vetor de pares ordenados com a informação das ligações entre os filtros. No final, é definido um vetor com o nome fgs de pares ordenados com a informação de cada *filter graph*, neste caso, um vetor constituído por apenas um par ordenado com o tipo de filtros do *filter graph* fg1 e as ligações entre eles.

Numa segunda fase, é chamado o *online partial evaluator* CApplicationInstantiator, que customiza alguns dos *template* artefactos desenvolvidos na fase do domínio da engenharia em componentes especializados e estabelece a ligação entre eles de forma a obter o código final da aplicação que satisfaz as necessidades do cliente. As linhas de código seguintes definem a classe *template* CApplicationInstantiator:

```
template<typename Config>
class CApplicationInstantiator
```

```
    : public CApplicationInstantiatorImpl<
          CIntelligentConnect::InstantiatorManager<Config::fgs>::type>,
          mpl::size<CIntelligentConnect::InstantiatorManager<Config::fgs>::type>::value>
{ };
```

A classe *template* CApplicationInstantiator recebe como parâmetro de entrada de *template* o tipo de dados Config, que contém a informação sobre as configurações do sistema definidas no ficheiro de configurações Config.h. A classe *template* CApplicationInstantiator herda da classe *template* CApplicationInstantiatorImpl, que recebe como parâmetros de entrada um vetor com a informação dos *filter graphs* da aplicação e o número de *filter graphs*. Convém salientar que o vetor fgs da estrutura Config não é utilizado diretamente como parâmetro de entrada da classe *template* CApplicationManager. Em vez disso, é utilizado o resultado da metafunção InstantiatorManager da estrutura CIntelligentConnect, que resolve os possíveis problemas de código desnecessário (*code bloat*) relacionados com instanciações diferentes do mesmo filtro no ato de compilação do código da aplicação.

A classe *template* CApplicationInstantiatorImpl implementa um conjunto de *templates* parcialmente especializadas que permitem criar os *filter graphs* da aplicação em função do seu número. Assim, são definidas *templates* parcialmente especializadas que permitem criar uma aplicação com apenas um *filter graph* e no máximo vinte. As linhas de código seguintes definem a classe *template* parcialmente especialziada CApplicationInstantiatorImpl no caso de o número de *filter graphs* da aplicação ser um:

```
template <typename FilterGraphTypes>
class CApplicationInstantiatorImpl<FilterGraphTypes, 1> {
    typedef typename mpl::at<FilterGraphTypes, mpl::long_<0> >::type item0;
    typedef typename mpl::first<item0>::type Filters0;
    typedef typename mpl::second<item0>::type Connections0;

    CFilterGraph<Filters0> _fg0;
public:
    void Initialize() {
        _fg0.ConnectFilters<Connections0>();
    }
    ...
};
```

As linhas de código seguintes apresentam a definição da classe *template* parcialmente especializada CApplicationInstantiatorImpl no caso de o número de *filter graphs* da aplicação serem vinte:

```
template <typename FilterGraphTypes>
class CApplicationInstantiatorImpl<FilterGraphTypes, 20> {
    typedef typename mpl::at<FilterGraphTypes, mpl::long_<0> >::type item0;
    ...
    typedef typename mpl::at<FilterGraphTypes, mpl::long_<19> >::type item19;
```

```
    typedef typename mpl::first<item0>::type Filters0;
    typedef typename mpl::second<item0>::type Connections0;
    ...
    typedef typename mpl::first<item19>::type Filters19;
    typedef typename mpl::second<item19>::type Connections19;

    CFilterGraph<Filters0> _fg0;
    ...
    CFilterGraph<Filters19> _fg19;
public:
    void Initialize() {
        _fg0.ConnectFilters<Connections0>();
        ...
        _fg19.ConnectFilters<Connections1>();
    }
    ...
};
```

Para gerar o código final da aplicação basta compilar as seguintes linhas de código:

```
#include <config.h>

int main(int argc, char *argv[])
{
    CApplicationInstantiator app<Config>;
    app.Initialize();
    app.Start();
    CUtils::Sleep(5);
    app.Stop();

    return 0;
}
```

6.3 *Model-Driven Template Metaprogramming*

A principal desvantagem de utilizar C++ *template metaprogramming*, para gerir a variabilidade do código de um sistema, prende-se com o facto da sintaxe do código TMP ser muito complexa e esotérica, comparativamente com o código convencional desenvolvido na linguagem de programação C++ (Cardoso et al., 2012g).

Para escrever código TMP, o programador necessita de:

i. Perceber um dos conceitos mais avançados da linguagem de programação C++, as *templates*;

ii. Pensar de forma diferente ao criar os novos algoritmos, visto que o código TMP está mais diretamente relacionado com programação funcional (lingua-

gem funcional), do que com a tradicional forma como se programa em C++ (linguagem imperativa).

6.3.1 *Template Markup Language* (TMPxML)

A linguagem TMPxML (*Template Markup Language*) foi desenvolvida no âmbito desta tese, para simplificar a forma como é escrito o código TMP. A linguagem TMPxML basea-se no formato XML e permite utilizar ferramentas *mainstream* para modelar, verificar e transformar código XML em código TMP.

A Figura 6.27, apresenta um diagrama com os passos necessários para criar código TMP, utilizando a linguagem TMPxML com o recurso a modelos:

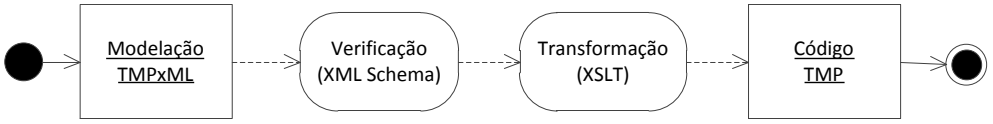


Figura 6.27: Geração automática de código TMP utilizando modelos através da linguagem TMPxML.

Tal como ilustra o diagrama da Figura 6.27, numa primeira fase, é feita a modelação dos blocos básicos necessários para implementar a funcionalidade pretendida, utilizando a linguagem TMPxML. De seguida, o código TMPxML é verificado e validado através de um ficheiro de *schema* XSD. Por fim, é utilizado um transformador XSLT para converter de forma automática o código TMPxML em código TMP.

No Anexo B, é apresentado o XML *Schema* da linguagem TMPxML, que permite fazer a validação de um documento escrito na linguagem TMPxML. O código do transformador XSLT, que faz a conversão automática do código escrito na linguagem TMPxML para a linguagem TMP, não é apresentado neste documento. O razão pende-se com o facto de o código do transformador ser demasiado extenso, fazendo com que o número de páginas da tese aumente de forma significativa.

Escrever Código TMP da *Framework* em TMPxML

A título de exemplo, é utilizada a linguagem linguagem TMPxML, para implementar o código TMP da *framework* generativa de videovigilância, que define os tipos de filtros e verifica se um filtro é de um determinado tipo.

Definição dos Tipos de Filtros Para definir os tipos de filtros da *framework* generativa de videovigilância, foram implementadas três *templates* em linguagem TMP: a *template* source_filter_t, que representa um filtro de entrada, a *template* transform_filter_t, que representa um filtro de transformação, e a *template* renderer_filter_t, que representa um filtro de saída.

O código seguinte define os tipos de filtros suportados pela *framework* generativa de videovigilância:

```
// estrutura que representa um filtro de entrada
struct source_filter_t {};

// estrutura que representa um filtro de transformação
struct transform_filter_t {};

// estrutura que representa um filtro de saída
struct renderer_filter_t {};
```

Para implementar o mesmo código utilizando TMPxML, são necessárias as seguintes linhas de código:

```
<!-- estrutura que representa um filtro de entrada -->
<template name="source_filter_t"/>

<!-- estrutura que representa um filtro de transformação -->
<template name="transform_filter_t"/>

<!-- estrutura que representa um filtro de saída -->
<template name="renderer_filter_t"/>
```

Metafunções para Validar os Tipos de Filtros Para verificar qual o tipo de um determinado filtro, foram definidas três metafunções em linguagem TMP: a metafunção is_source_filter, que devolve verdade se um determinado filtro é do tipo filtro de entrada, a metafunção is_transform_filter, que devolve verdade se um determinado filtro é do tipo filtro de transformação, e a metafunção is_renderer_filter, que devolve verdade se um determinado filtro é do tipo filtro de saída.

O código seguinte define as três metafunções para validação dos tipos de filtros, suportados pela *framework* generativa de videovigilância:

```
// metafunção que verifica se um filtro é do tipo filtro de entrada
template <typename FilterType>
struct is_source_filter
   : public is_same<typename FilterType::type, source_filter_t>
{};

// metafunção que verifica se um filtro é do tipo filtro de transformação
template <typename FilterType>
```

```
struct is_transform_filter
    : public is_same<typename FilterType::type, transform_filter_t>
{};

// metafunção que verifica se um filtro é do tipo filtro de saída
template <typename FilterType>
struct is_renderer_filter
    : public is_same<typename FilterType::type, renderer_filter_t>
{};
```

Para implementar o mesmo código utilizando TMPxML, são necessárias as seguintes linhas de código:

```
<!-- metafunção que verifica se um filtro é do tipo filtro de entrada -->
<template name="is_source_filter"/>
    <parameter name="FilterType"/>
    <extension type="is_same">
        <parameter value="FilterType::type"/>
        <parameter value="source_filter_t"/>
    </extension>
</template>

<!-- metafunção que verifica se um filtro é do tipo filtro de transformação -->
<template name="is_transform_filter"/>
    <parameter name="FilterType"/>
    <extension type="is_same">
        <parameter value="FilterType::type"/>
        <parameter value="transform_filter_t"/>
    </extension>
</template>

<!-- metafunção que verifica se um filtro é do tipo filtro de saída -->
<template name="is_renderer_filter"/>
    <parameter name="FilterType"/>
    <extension type="is_same">
        <parameter value="FilterType::type"/>
        <parameter value="renderer_filter_t"/>
    </extension>
</template>
```

Ferramenta de Modelação TMPxML

A ferramenta de modelação TMPxML, é um *plugin* para Eclipse que permite criar código TMP utilizando modelos. Ao criar os modelos na ferramenta de modelação TMPxML, estes são gravados em linguagem TMPxML, que depois de validada, é transformada em linguagem TMP, utilizando um transformador XSLT.

O *plugin* de modelação TMPxML para Eclipse, foi desenvolvido com a *Eclipse Modeling Framework* (EMF), utilizando o XML *Schema* da linguagem TMPxML como entrada para criar os modelos EMF.

A Figura 6.28, apresenta um imagem do ambiente de desenvolvimento da ferramenta de modelação TMPxML para Eclipse:

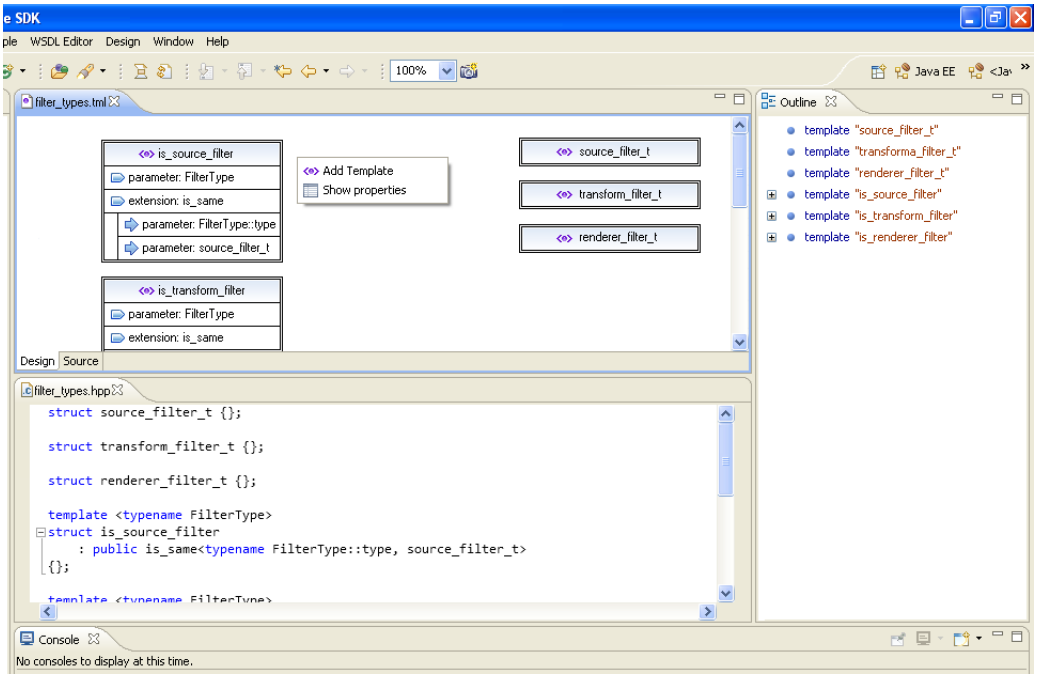


Figura 6.28: Ambiente de desenvolvimento da ferramenta de modelação TMPxML para Eclipse.

O ambiente de desenvolvimento da ferramenta de modelação TMPxML, é composto por dois editores: o editor de modelação TMPxML (vista superior) e o editor de código TMP (vista inferior). O editor de modelação TMPxML permite criar/editar *templates* de duas formas: utilizando modelos gráficos (página de desenho) ou editando código TMPxML manualmente (página de código). O editor de código TMP apresenta o código TMP, que é gerado de forma automática pelo transformador XSLT, de código TMPxML para código TMP. Finalmente, na vista *Outline* é possível visualizar as *templates* criadas e as suas propriedades de forma hierárquica.

Ao utilizar a ferramenta de modelação TMPxML, o programador desenvolve código TMP de uma forma mais rápida e eficaz. O uso de modelos, facilita a compreensão do código e aumenta a produtividade/manutenção do código TMP, abstraindo o programador da complexidade inerente ao desenvolvimento de código TMP.

6.4 Conclusão

Neste capítulo foi apresentado o processo de análise, desenho e implementação de uma *framework* generativa de videovigilância. A integração de técnicas MDD, *Agile*, SPL e programação generativa permitiram criar a *framework* com o intuito de satisfazer os vários requisitos impostos no desenvolvimento de sistemas de videovigilância de última geração.

Durante o processo de análise, desenho e implementação da *framework* generativa, foi seguida uma abordagem mais inovadora do que a abordagem que normalmente os fabricantes deste tipo de sistemas adotam, o desenvolvimento de sistemas baseado em técnicas *ad-hoc* e focado num só produto da videovigilância. Assim, seguindo as técnicas de desenvolvimento SPL, os sistemas de videovigilância são vistos como uma família de produtos que partilham funcionalidades comuns a todas as configurações possíveis, e onde só os pontos de variação necessitam de ser implementados. Para obter uma determinada configuração, basta selecionar os componentes necessários de um repositório de componentes e estabelecer a ligação entre eles, e a *framework* gera de forma automática o código otimizado para o fim proposto.

Neste capítulo, também foi apresentada a ferramenta de modelação TMPxML. A ferramenta de modelação TMPxML, permite gerar de forma automática, código TMP a partir de modelos TMPxML. Ao utilizar modelos TMPxML, o programador cria código TMP de uma forma mais rápida e eficaz, comparativamente com o método manual de desenvolvimento de código TMP.

A convergência de sinergias destas tecnologias, apresenta como principais vantagens: (1) o desenvolvimento de produtos com maior qualidade, (2) redução do *time-to-market* e (3) a facilidade de expandir o âmbito do domínio da *framework* com a introdução de novas funcionalidades.

Uma das vantagens da *framework* generativa desenvolvida neste capítulo relativamente a outras é a sua simplicidade e o elevado nível de integração com ferramentas *mainstream* e tecnologias, tais como, XML, XSLT, C++ *template metaprogramming*, programação generativa, MDD, *Agile*, e SPL.

Capítulo 7

Resultados Experimentais

7.1 Introdução

No capítulo anterior, foi apresentada e discutida a *framework* generativa de videovigilância. Essencialmente, foram apresentados os dois estágios do *Software Product Line*: o *domínio de engenharia* e a *engenharia da aplicação*. O *domínio de engenharia* permite analisar, desenhar e implementar a *framework* generativa de videovigilância. A *engenharia da aplicação* permite selecionar a partir de um repositório de componentes, os componentes necessários para obter a configuração do sistema pretendida.

Neste capítulo, são apresentados os resultados experimentais dos testes realizados ao nível das características não funcionais e da gestão do código fonte do sistema, em três plataformas com diferentes características de *hardware* e *software*. Os testes das características não funcionais e gestão do código fonte do sistema foram realizados em três implementações diferentes do sistema: uma implementação desenvolvida em linguagem de programação C e duas implementações desenvolvidas em linguagem de programação C++. Na implementação em linguagem de programação C, a variabilidade do sistema é gerida com o recurso a apontadores para funções e compilação condicional. Numa das implementações em linguagem de programação C++, a variabilidade do sistema é gerida com polimorfismo dinâmico e compilação condicional. Na outra implementação em linguagem de programação C++ (TMP), é utilizado o polimorfismo paramétrico e *template metaprogramming* para gerir a variabilidade do sistema.

7.2 Ambiente de Testes

Para uma melhor validação dos resultados experimentais, os testes efetuados foram realizados em arquiteturas/plataformas com diferentes características de *hardware* e *software*. A tabela 7.1, apresenta as características de cada uma das máquinas de testes:

Tabela 7.1: Características de *hardware* e *software* das máquinas de testes.

Máquina		Características
PC (PC)		
	Hardware	Arquitetura: x86
		Processador: Intel(R) Core(TM)2 Duo
		Velocidade do CPU: 2,66 GHz
		RAM: 2,83 GB
		Disco: 200 GB
	Software	Sistema Operativo: Ubuntu 9.10
		Compilador: gcc versão 4.4.1
BeagleBoard-xM (BB)		
	Hardware	Arquitetura: ARM
		Processador: TI DM3730 Super-scalar ARM Cortex TM -A8
		Velocidade do CPU: 1 GHz
		DSP: TMS320C64x 520 MHz
		RAM: 512 MB LPDDR
		Cartão microSD: 4 GB
	Software	Sistema Operativo: Angstrom 2010.7
		Compilador: arm-linux-gcc (GCC) 4.2.2
Fonera FON2100A (FN)		
	Hardware	Arquitetura: MIPS 4KEc
		Processador: Atheros AR2315
		Velocidade do CPU: 183 MHz
		RAM: 16 MB
		Flash: 8 MB
	Software	Sistema Operativo: OpenWrt Kamikaze 8.09.2
		Compilador: mips-linux-gcc versão 4.1.2

Tal como ilustra a tabela 7.1, os testes foram realizados em três plataformas com características de *hardware* diferentes: PC (PC), BeagleBoard-xM (BB) e Fonera FON2100A (FN). A máquina denominada PC, é uma máquina genérica (Jacob et al., 2007) com um processador Intel - arquitetura x86. A plataforma denominada BB, é uma *appropriately built machine* (Jacob et al., 2007) com um processador da Texas Instruments - arquitetura ARM - e um DSP (*Digital Signal Processor*) da Texas Instruments especializado em processamento digital de sinal, utilizado para processar sinais de áudio e vídeo. Por fim, a plataforma denominada FN, é uma *appropriately built machine* com um processador da Atheros - arquitetura MIPS.

Nas três plataformas de testes foi instalado o sistema operativo Linux. Na plataforma PC foi instalada uma distribuição normal do Linux (Ubuntu). Nas outras duas plataformas, foi instalado o Linux Embebido (Angstrom e OpenWrt). Para compilar cada um dos filtros da *framework* generativa de videovigilância, foi utilizado o compilador de C/C++ da GNU (gcc/g++). No ato de compilação foi definida a opção de compilação *-O2*, para obter um ficheiro executável final com um bom balanceamento entre as métricas de desempenho e o tamanho do executável.

7.2.1 Ferramentas Utilizadas

Para obter os resultados relacionados com os testes das características não funcionais foram utilizados dois utilitários do sistema operativo Linux: o utilitário *time* e o utilitário *size*. O utilitário *time* apresenta o tempo de execução em segundos de um determinado executável. O utilitário *size* apresenta o tamanho do executável gerado pelo compilador em *bytes*.

Finalmente, para obter os resultados relacionados com as métricas de facilidade de gestão e expansão do código do sistema, utilizou-se o programa *Understand* (scitools). O programa *Understand* é uma ferramenta de análise estática de métricas relacionadas com a análise e manutenção de código fonte de programas desenvolvidos em várias linguagens de programação.

7.3 Métricas de Teste

As métricas utilizadas para fazer os testes comparativos entre as três implementações, assentam essencialmente em métricas de testes relacionadas com as características não funcionais do sistema e a facilidade de gestão/expansão do código do sistema.

Para realizar os testes das características não funcionais do sistema para as três plataformas de testes, foram adquiridos o tempo de execução de cada uma das implementações bem como o tamanho do ficheiro executável gerado.

Para verificar o grau de complexidade inerente à gestão/expansão do código do sistema, são analisadas as seguintes métricas:

- *Linhas de Código (LOC)*: conta o número de linhas de código - sem comentários e linhas em branco - de um produto gerado da *framework* generativa de videovigilância;

- *Número de Packages (NOP)*: conta o número de *packages* presentes num determinado produto. No caso da implementação ser na linguagem de programação C, considera-se um *package* um ficheiro de código com a extensão .c. No caso da implementação ser em linguagem de programação C++, considera-se um *package* um *namespace*;

- *Número de Classes (NOC)*: conta o número de classes presentes num determinado produto gerado da *framework* generativa de videovigilância;

7.4 Testes Realizados

Como a *framework* generativa de videovigilância é constituída por vários filtros que podem ser configurados e interligados de variadas formas, a realização dos testes e apresentação dos resultados tornou-se uma tarefa complexa.

Para simplificar o processo sem por em causa a veracidade dos resultados obtidos, em vez de se apresentar os testes em todos filtros e para todas as configurações possíveis do *filter graph*, decidiu-se apresentar apenas os resultados efetuados num dos filtros.

De forma a que o filtro selecionado possa representar a maioria dos filtros ao nível dos testes e dos resultados obtidos, bem como as conclusões a retirar dos testes comparativos entre as várias soluções de implementação da variabilidade do sistema, na seleção do filtro obedeceu-se aos seguintes requisitos:

i. O filtro selecionado deve implementar uma das funcionalidades mais utilizadas em sistemas de videovigilância;

ii. O filtro deve apresentar um grau de variabilidade médio, isto é, não deve ser um dos filtros com poucas funcionalidades variáveis, nem um dos filtros com muitas funcionalidades variáveis. Desta forma, o filtro representará, em média, todos os filtros da *framework* generativa de videovigilância;

iii. O filtro deve utilizar como base de implementação uma biblioteca externa com o código fonte disponível, de modo a que a comunidade científica possa

reproduzir os resultados apresentados;

iv. O filtro deve ter como base de implementação uma biblioteca externa implementada em linguagem de programação C. Assim, são apresentados todos os passos necessários para converter o código *legacy* da biblioteca desenvolvido em linguagem de programação C para código desenvolvido em linguagem de programação C++, utilizando polimorfismo paramétrico e *template metaprogramming* para implementar as funcionalidades variáveis;

v. O filtro deve apresentar todos os tipos de funcionalidades: funcionalidades obrigatórias, funcionalidades opcionais, funcionalidades alternativas e funcionalidades acumulativas.

Como um dos objetivos do plano de testes é também a realização de testes em várias configurações do *filter graph*, ao selecionar o filtro também se teve em consideração os seguintes requisitos:

i. O filtro deve ser composto por vários estágios de processamento seguindo o modelo de execução em *pipeline*, tal como acontece com o *filter graph*;

ii. Um dos métodos da gestão das funcionalidades variáveis do filtro, deve ser implementado na linguagem de programação C++ utilizando polimorfismo dinâmico, recorrendo ao uso de herança e funções virtuais. Desta forma, é recriado um *pipeline* baseado na mesma filosofia de implementação do *filter graph* do *DirectShow*.

Depois de analisar todos os filtros da *framework* generativa de videovigilância, chegou-se à conclusão que o filtro que melhor satisfaz os requisitos impostos anteriormente, é o filtro de transformação CJPEGDecTransform, que faz a descompressão de imagens no formato JPEG para um formato de imagem descomprimido (*raw*).

7.5 Filtro de Transformação *CJPEGDecTransform*

O filtro de transformação CJPEGDecTransform recebe à entrada uma imagem no formato JPEG e faz a descompressão da mesma para um formato descomprimido, neste caso específico, para o formato BMP. Na *framework* generativa de videovigilância utilizou-se como base de implementação do filtro, a mesma biblioteca que

o *DirectShow* utiliza para fazer a descompressão de imagens no formato JPEG - a biblioteca *libjpeg* (Lane). A biblioteca *libjpeg* foi desenvolvida por Tom Lane e o *Independent JPEG Group* (IJG) durante os anos 90. Hoje em dia quem faz a manutenção da biblioteca é um grupo de programadores da comunidade *SourceForge*.

A biblioteca é totalmente desenvolvida na linguagem de programação C. Para implementar cada um dos estágios de processamento descrito no *standard* JPEG são utilizados diferentes módulos, os quais são inicializados na função main através de apontadores para funções.

Em termos de configurações, a biblioteca *libjpeg* utiliza algumas diretivas do preprocessador C (compilação condicional) que permitem definir quais as funcionalidades que são compiladas. Embora seja um ponto de partida para que o código da biblioteca seja totalmente configurável, fica muito aquém das expectativas, visto que muitas das funcionalidades não são configuráveis.

7.5.1 Extração das Funcionalidades

Tal como foi referido anteriormente, não é possível configurar muitas das funcionalidades variáveis da biblioteca *libjpeg*. Assim, para que seja possível tornar a biblioteca totalmente configurável foram realizados três passos: (i) numa primeira fase utilizaram-se diretivas do preprocessador C (compilação condicional) para delimitar o código associado a cada uma delas. (ii) De seguida, o código em linguagem de programação C foi convertido para código objeto em linguagem de programação C++, sendo utilizado o polimorfismo dinâmico e compilação condicional para gerir a variabilidade. (iii) As funcionalidades variáveis foram implementadas em linguagem de programação C++, utilizando polimorfismo paramétrico (*templates* especializadas) e *template metaprogramming*.

A tabela 7.2 apresenta as funcionalidades variáveis identificadas na biblioteca *libjpeg* e uma breve descrição de cada uma delas:

A biblioteca *libjpeg*, apresenta oito funcionalidades variáveis: *output*, *colorspace*, *buffer*, *coef*, *entropy*, *idct*, *post* e *upsampling*. Duas delas, são funcionalidades que permitem gerir o fluxo de execução dos dados - *buffer* e *post*. As restantes funcionalidades, implementam operações de descompressão descritas no *standard* JPEG.

A tabela 7.3 apresenta os módulos em linguagem de programação C, que imple-

Tabela 7.2: Funcionalidades variáveis identificadas na biblioteca *libjpeg*

Funcionalidade	Descrição
output	implementa funções relacionadas com os formatos de imagem de saída.
colorspace	implementa funções de conversão do espaço de cor.
buffer	implementa funções para a gestão de memória do *buffer* de dados.
coef	implementa funções para o cálculo dos coeficientes de quantização.
entropy	implementa funções para a descompressão dos *bits* do código entrópico.
idct	implementa funções para o calculo da transformada inversa do co-seno.
post	implementa funções de pós-processamento.
upsampling	implementa funções para efetuar o *upsampling*.

mentam cada uma das funcionalidades variáveis da biblioteca *libjpeg*:

Tabela 7.3: Módulos que implementam cada uma das funcionalidades variáveis da biblioteca *libjpeg*

Funcionalidade	Módulos
output	wrbmp.c, wrgif.c, wrppm.c, wrrle.c, wrtarga.c e rdcolmap.c
colorspace	jdcolor.c
buffer	jdmainct.c
coef	jdcoefct.c
entropy	jidctflt.c, jidctfst.c e jidctint.c
idct	jdhuff.c, jdarith.c e jddctmgr.c
post	jdpostct.c
upsampling	jdsample.c e jdmerge.c

Do mesmo modo que o *filter graph* do *DirectShow* é constituído por um conjunto de filtros, que estão ligados entre si, o fluxo de execução de dados da biblioteca *libjpeg* também pode ser visto como um *filter graph* que é constituído por um conjunto de módulos (filtros) que estão ligados entre si. Os módulos estão ligados uns aos outros através de apontadores para funções que são inicializados em função das características da imagem JPEG a descomprimir. A figura 7.1 ilustra cada um desses módulos de processamento e a ligação entre eles.

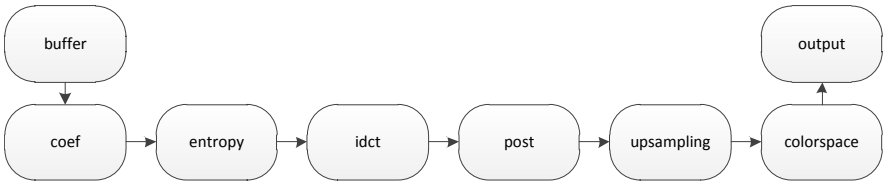


Figura 7.1: Ligação entre os módulos da biblioteca *libjpeg*.

O módulo *buffer* armazena os dados da imagem no formato JPEG. De seguida, os módulos *coef* e *entropy*, calculam os coeficientes de quantização e fazem a descodificação do código de *bits* entrópico para *bytes*, respetivamente. O módulo *idct*

aplica a transformada inversa do co-seno a cada uma da amostras com o tamanho de 8x8 *bytes*. Se for necessário, são executadas funções de pós-processamento no módulo *post*. O módulo *upsampling* calcula as amostras da crominância para que a imagem tenha o mesmo número de amostras da componente de luminância e das duas componentes de crominância. Por fim, é feita a conversão do formato de cor YCbCr para o formato de cor RGB, para que a imagem seja gravada no formato RGB.

Como em SPL, as funcionalidades variáveis de um sistema são representadas através do diagrama de funcionalidades, a figura 7.2 apresenta o diagrama de funcionalidades da biblioteca *libjpeg*.

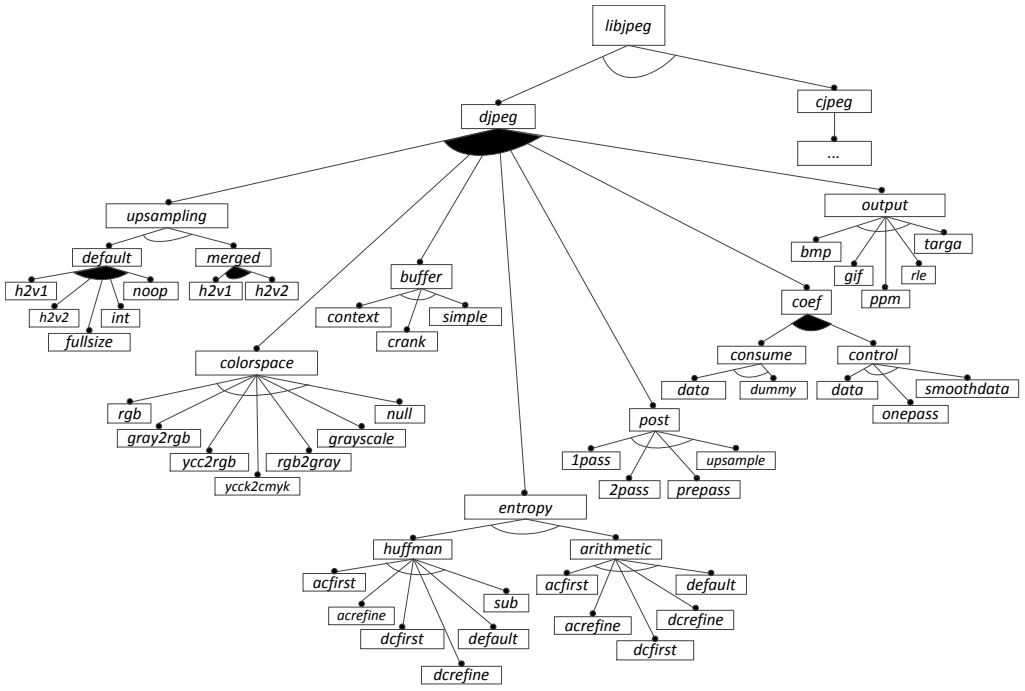


Figura 7.2: Diagrama de funcionalidades da biblioteca *libjpeg*.

7.5.2 Testes das Características Não Funcionais

Os testes das características não funcionais são divididos em dois tipos de testes: os testes relacionados com o tempo de execução e os testes que permitem obter o tamanho do ficheiro executável. Os testes relacionados com o tempo de execução, permitem obter os tempos necessários para executar cada uma das implementações do filtro (C, C++ e TMP), nas três máquinas de teste da tabela 7.1. Os testes relacionados com o tamanho do ficheiro permitem obter o tamanho dos ficheiros executáveis para cada uma das implementações do filtro (C, C++ e TMP).

Tempo de execução

Para obter os tempos de execução de cada uma das implementações do filtro, foram executados os ficheiros binários de cada uma destas, utilizando o utilitário *time*, para a descompressão de 25 imagens com a resolução de 320x240 do formato de imagem JPEG para o formato de imagem BMP. Como nas implementações do filtro em C e C++, o filtro pode ser compilado com todas as funcionalidades selecionadas (versão não otimizada) ou compilado apenas com as funcionalidades necessárias para fazer a descompressão da imagem JPEG em questão (versão otimizada), são apresentadas duas secções com os testes comparativos das duas implementações com a implementação em TMP.

A implementação TMP do filtro é sempre compilada na versão otimizada, visto ser utilizado o polimorfismo paramétrico para a gestão da variabilidade do filtro.

Versão Não Otimizada O gráfico da figura 7.3 apresenta os resultados do tempo de execução das implementações C e C++ não otimizadas e da implementação TMP, nas três máquinas de testes:

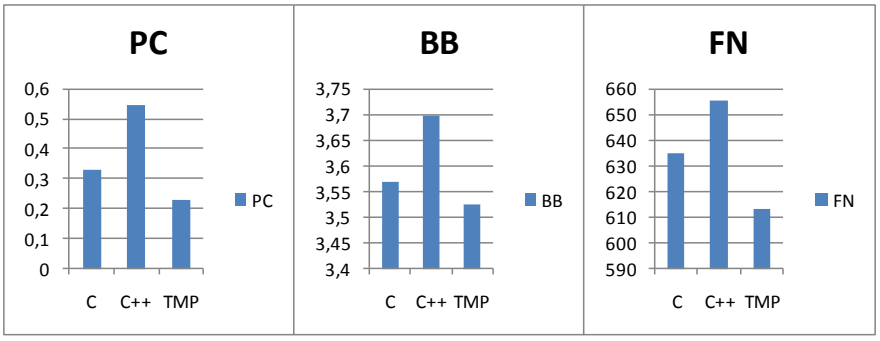


Figura 7.3: Tempo de execução com todas as funcionalidades selecionadas para a implementação em linguagem de programação C e C++.

Tal como o gráfico da figura 7.3 ilustra, a implementação TMP apresenta sempre tempos de execução menores do que as outras duas implementações (C e C++). A implementação TMP apresenta melhores resultados porque o código executável é totalmente otimizado para o fim proposto. Como as versões C e C++ foram compiladas com todas as funcionalidades selecionadas, a linearidade do código é afetada devido ao elevado número de instruções que não são executadas. Assim, é produzido um impacto negativo na performance do sistema devido ao elevado número de saltos no código que conduz ao aumento da probabilidade da ocorrência de *cache misses* e consequentemente à diminuição do desempenho.

Ao analisar cada uma das implementações da biblioteca nas diferentes máquinas de testes, é notório que quanto menos forem os recursos das plataformas de testes, maior é a diferença entre o tempo de execução das versões não otimizadas C e C++ e a versão TMP. Logo, sempre que o sistema seja desenvolvido para ser executado num sistema com poucos recursos, tais como, os sistemas embebidos, é necessário ter em consideração a forma como é feita a gestão da variabilidade do código do sistema.

Versão Otimizada O gráfico da figura 7.4, apresenta os resultados das mesmas simulações do gráfico da figura 7.3, mas com versões de C e C++ otimizadas. Isto é, nas versões C e C++ só foram selecionadas as funcionalidades necessárias para a descompressão das imagens JPEG.

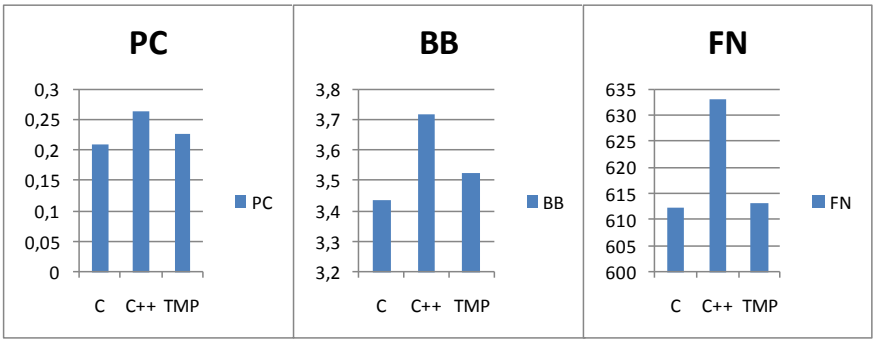


Figura 7.4: Tempo de execução com as funcionalidades necessárias selecionadas.

Neste caso, a implementação C otimizada apresenta sempre tempos de execução menores do que as outras duas versões (C++ e TMP). Desta vez, como o código das implementações C e C++ é otimizado, o código executável das duas implementações é mais linear, fazendo com que o tempo de execução diminua.

A implementação em C apresenta melhores resultados que a implementação TMP devido, essencialmente, as dois fatores: (i) embora a implementação TMP seja otimizada, existem alguns pontos de variabilidade que não são tão otimizados como os implementados através de apontadores para funções em C. (ii) Alguns dos objetos na implementação TMP serem alocados dinamicamente recorrendo à função `new` (a alocação dinâmica tem um impacto negativo no desempenho).

O tempo de execução da implementação C++ apresenta sempre piores resultados que as outras duas implementações. Isso deve-se ao facto de o polimorfismo dinâmico (funções virtuais) adicionarem um nível de indirecção nas chamada das funções virtuais, devido ao facto de as funções serem resolvidas em tempo de execução

e também devido ao facto de todos os objetos que implementam funcionalidades variáveis serem alocados dinamicamente através da função new.

Tamanho do Ficheiro Executável

A tabela 7.4 apresenta o número de *bytes* que a biblioteca *libjpeg* ocupa, para as funcionalidades comuns e as funcionalidades variáveis:

Tabela 7.4: Tamanho em *bytes* das funcionalidades comuns e variáveis da biblioteca *libjpeg*.

Funcionalidades	Número de *bytes*	Percentagem
comuns	50.271	36,38%
variáveis	87.898	63,62%

Dos 138.169 *bytes* que implementam as funcionalidades comuns e variáveis da biblioteca, mais de 60% do número de *bytes* implementam as funcionalidades variáveis. Isto é, mais de metade do código binário resultante da compilação da biblioteca *libjpeg* implementa funcionalidades variáveis. Logo, é necessário ter especial atenção na forma como é feita a gestão do código que implementa as funcionalidades variáveis, de forma a que o código binário final da biblioteca seja o mais otimizável possível para o fim proposto, não comprometendo assim, o desempenho do sistema.

O gráfico da figura 7.5 apresenta o número de *bytes* das funcionalidades variáveis em função de cada um dos módulos da tabela 7.2:

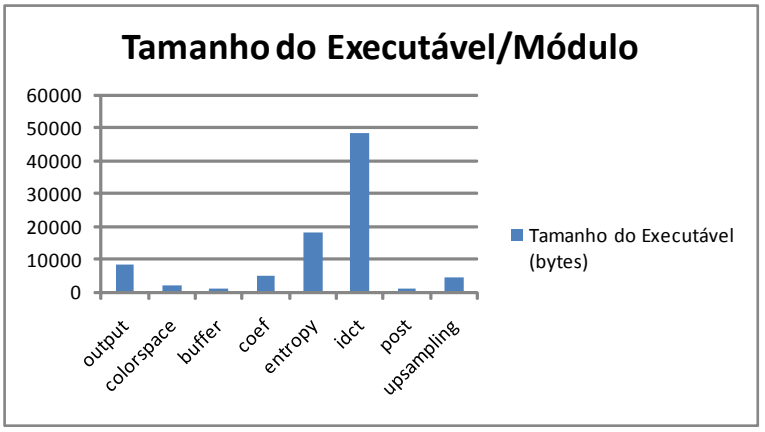


Figura 7.5: Número de *bytes* de cada uma das funcionalidades variáveis.

A funcionalidade variável que ocupa mais *bytes* no tamanho do ficheiro executável,

é a funcionalidade *idct*. A funcionalidade *idct* é uma funcionalidade obrigatória e define 35 sub-funcionalidades alternativas, que permitem executar a transformada inversa do co-seno.

De seguida, a funcionalidade que mais *bytes* ocupa no ficheiro executável da biblioteca *libjpeg*, é a funcionalidade *entropy*. A funcionalidade *entropy* faz a conversão dos *bits* de cada uma das amostras para *bytes* (descodificação *huffman* ou descodificação aritmética). Tal como a funcionalidade *idct*, a funcionalidade *entropy*, também é uma funcionalidade obrigatória. Esta, é constituída por duas sub-funcionalidades alternativas: a sub-funcionalidade *huffman* e a sub-funcionalidade *arithmetic*. A sub-funcionalidade *huffman* define 6 métodos alternativos para a descodificação *huffman*. A sub-funcionalidade *arithmetic* define 5 métodos alternativos para a descodificação aritmética.

Versão Não Otimizada O gráfico da figura 7.6, apresenta o tamanho em *bytes* que cada um dos módulos ocupa para cada uma das implementações da biblioteca *libjpeg* (C, C++ e TMP). Nas implementações C e C++ são selecionadas todas as funcionalidades variáveis.

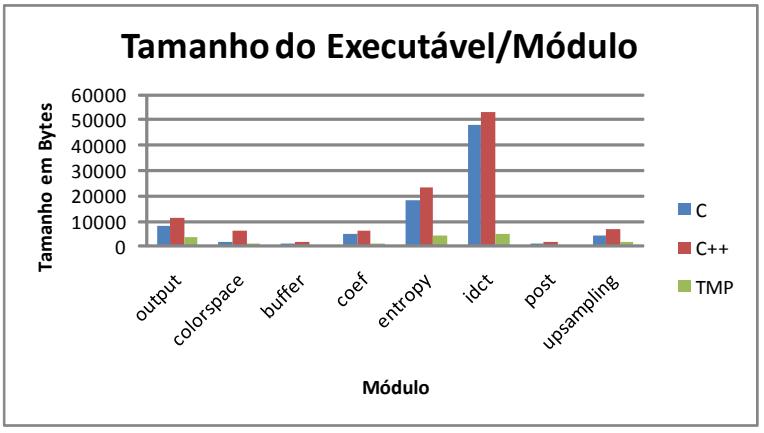


Figura 7.6: Número de *bytes* que cada um dos módulos ocupa para cada uma das implementações da biblioteca *libjpeg*.

Do gráfico conclui-se que nas implementações não otimizadas em C e C++, todos os módulos da biblioteca *libjpeg* ocupam um número significativo de *bytes* comparativamente com a implementação TMP.

Versão Otimizada O gráfico da figura 7.7, apresenta o tamanho em *bytes* que cada um dos módulos ocupa para cada uma das implementações otimizadas da

biblioteca *libjpeg* (C, C++ e TMP).

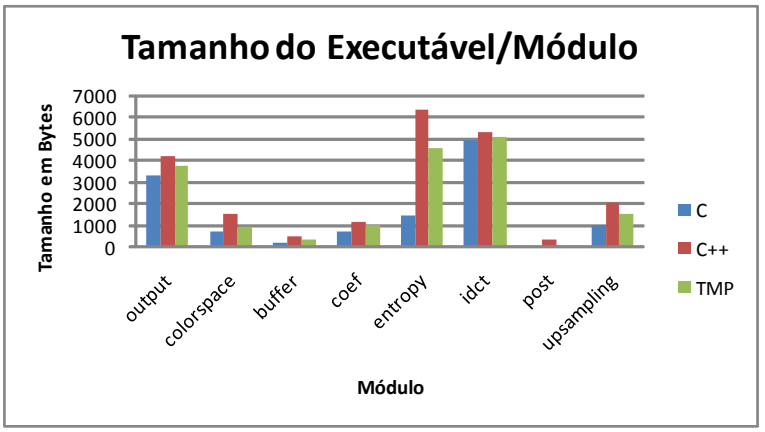


Figura 7.7: Número de *bytes* que cada um dos módulos ocupa nas versões otimizadas das implementações da biblioteca *libjpeg*.

Neste caso, e ao contrário da versão não otimizada, todos os módulos na implementação em C ocupam menos *bytes* que as duas implementações em C++ e TMP. De realçar que a implementação em TMP ocupa menos *bytes* que a implementação em C++.

7.5.3 Testes de Gestão do Código

Embora os testes relacionados com o desempenho do sistema sejam de especial importância em tempo de execução, não menos importante é a forma como e feita a gestão da variabilidade do código da biblioteca *libjpeg*. Assim, nesta secção o que se pretende é fazer testes comparativos entre as três implementações da biblioteca *libjpeg* ao nível da gestão do código, mais propriamente, ao nível das métricas LOC, NOP e NOC.

O gráfico da figura 7.5 apresenta os valores de cada uma das métricas em cada um dos módulos, para as diferentes implementações da biblioteca *libjpeg*. Do gráfico, conclui-se que o número de linhas de código (LOC) das três implementações é praticamente o mesmo. No que diz respeito às métricas relacionadas com o número de *packages* (NOP) e o número de classes (NOC), a implementação C apresenta valores inferiores às implementações em C++ e TMP. Isto é, as implementações em C++ e TMP apresentam um código mais modelar e com um nível de encapsulamento de cada uma das funcionalidades superior à versão C.

A implementação em TMP destaca-se das outras duas implementações, porque a

Tabela 7.5: Valores de cada uma das métricas em cada um dos módulos para as diferentes implementações da biblioteca *libjpeg*

	LOC			*NOP*			*NOC*		
Funcionalidade	*C*	*C++*	*TMP*	*C*	*C++*	*TMP*	*C*	*C++*	*TMP*
output	1159	974	790	6	6	6	7	7	7
colorspace	302	412	405	1	8	8	9	9	9
buffer	248	100	87	1	3	3	4	4	5
coef	477	523	372	1	6	6	8	8	9
entropy	3737	1535	1838	3	13	13	16	16	19
idct	1571	4181	4014	3	35	35	36	36	36
post	168	190	98	1	5	5	6	6	6
upsampling	434	562	302	2	9	9	12	12	15
common	4269	3775	4074	15	11	11	7	7	10
total	**12365**	**12252**	**11980**	**33**	**96**	**96**	**105**	**105**	**116**

gestão da variabilidade do código é feita pelo compilador, através do polimorfismo paramétrico e do uso de *template metaprogramming*. Nas implementações em C e C++ é utilizada compilação condicional, que polui o código fonte da biblioteca *libjpeg*, sendo mais difícil a sua leitura e mais suscetível a erros. Sempre que seja necessário introduzir novas funcionalidades à biblioteca, nas versões em C e C++ é necessário alterar os *managers* das funcionalidades, por forma a que estas possam ser instanciadas. Na versão TMP não é necessário, porque é o próprio compilador que gere as instâncias de cada funcionalidade variável do sistema.

7.6 Conclusão

Neste capítulo foram apresentadas os resultados experimentais de três implementações diferentes do mesmo filtro (C, C++ e TMP), realizados em três plataformas com diferentes características de *hardware* e *software*. Os testes realizados foram divididos em dois tipos de testes: os testes das características não funcionais e os testes de gestão do código do filtro.

Dos testes realizados ao nível do desempenho nas três implementações do filtro, verifica-se que em versões não otimizadas do filtro (C e C++), o tempo de execução e o tamanho do executável são muito superiores ao tempo de execução e ao tamanho do executável da implementação em TMP. Por sua vez, quando são utilizadas versões otimizadas de C e C++, a implementação em C necessita de menos tempo de execução e número de *bytes* no executável do que as outras duas implementações (C e TMP). Embora, os resultados das implementações TMP e C sejam muito parecidos, as diferenças acentuam-se em sistemas com poucos recursos, tais

como, os sistemas embebidos.

No que diz respeito às métricas de gestão do código, a implementação em TMP apresenta mais flexibilidade ao nível da gestão do código e expansão do mesmo. Um dos fatores que contribui para que isso aconteça, é o maior número de classes (NOC) que esta implementação apresenta em relação às outras duas implementações, o maior número de *packages* e o menor número de linhas de código das funcionalidades variáveis. Um outro fator que permite uma maior flexibilidade ao nível da gestão do código e expansão do mesmo na implementação TMP, é que a gestão da variabilidade é feita em tempo de compilação utilizando polimorfismo paramétrico e *template metaprogramming*, ao contrário das outras duas implementações onde a variabilidade é resolvida em tempo de execução. Como os módulos necessários para uma determinada configuração na implementação TMP são geridos pelo próprio compilador através de *templates* especializadas, esta implementação apresenta um vantagem significativa em termos da gestão das funcionalidades variáveis do sistema, visto que as outras usam a compilação condicional com todos os problemas associados ao efeito "ifdef hell"(Favre, 1997).

Assim, a a título de conclusão, a versão em TMP permite resolver os dois problemas associados aos sistemas de videovigilância atuais: o desempenho e a variabilidade. Isto é, com a implementação TMP é possível obter um melhor compromisso entre estas duas grandezas.

Capítulo 8

Conclusão e Trabalho Futuro

Neste capítulo, são apresentadas as principais conclusões do trabalho desenvolvido nesta tese de doutoramento. De seguida, são propostas possíveis linhas de investigação a serem realizadas como trabalho futuro.

8.1 Conclusão

O aumento da criminalidade e a possibilidade da ocorrências de eventos anómalos, que possam pôr em causa a segurança das pessoas e os seus bens, fizeram com que o número de sistemas de videovigilância, instalados em espaços públicos e privados, aumentasse de forma considerável. O aumento do número de sistemas de videovigilância instalados deve-se, essencialmente, a dois fatores, que estão diretamente relacionados com as funções que estes desempenham enquanto agentes de segurança: (i) os sistemas de videovigilância agem de forma automática, perante a deteção de um evento anómalo (agente proactivo) e (ii) os sistemas de videovigilância permitem a consulta dos eventos anómalos registados, mesmo após a sua ocorrência (agente reativo).

A necessidade de instalar sistemas de videovigilância, em espaços públicos e privados, levou a comunidade científica e a indústria, a dar uma especial atenção à forma como estes são projetados. A maioria dos atuais sistemas de videovigilância são projetados de forma *ad hoc*, fazendo com que estes não possam ser utilizados em diferentes contextos e em ambientes distribuídos. As arquiteturas centralizadas que os caracterizam, fazem com que a sua integração com outro tipo de sistemas,

se torne uma tarefa complexa ou até, em muitos casos, impossível.

Recentemente, a comunidade científica começou a considerar o uso de arquiteturas para sistemas de videovigilância inteligentes, sob a forma de *middlewares* e *frameworks* especializadas, baseadas em paradigmas multiagentes. Embora esta nova abordagem permita gerir a complexidade e a heterogeneidade dos sistemas de videovigilância, produzem um impacto negativo no desempenho do sistema. Isso deve-se, essencialmente, ao facto de serem utilizadas técnicas de programação dinâmicas para gerir a variabilidade do código do sistema. Desta forma, não é possível em tempo de compilação, compilar apenas as funcionalidades necessárias para a configuração do sistema pretendido, fazendo com que o código do sistema não esteja otimizado para o fim proposto, e afete negativamente o desempenho e o *footprint* do mesmo.

O trabalho apresentado nesta tese de doutoramento, visa o desenvolvimento de uma *framework* generativa de videovigilância, totalmente reconfigurável. A *framework* é capaz de gerir, de forma eficaz, o elevado grau de variabilidade que caracterizam os sistemas de videovigilância, sem por em causa o desempenho dos mesmos. Assim, ao utilizar um ambiente de desenvolvimento de sistemas de videovigilância inteligentes, que utiliza o perfil da aplicação para guiar o processo de desenvolvimento da mesmo, no ato da compilação é gerado um ficheiro executável totalmente otimizado para a aplicação alvo.

Ao desenvolver a *framework* generativa de videovigilância, foram utilizadas combinações de técnicas e tecnologia de *Software Product Line* (SPL), *Model Driven Development* (MDD) e *Agile*, para gerir de forma simples e eficaz a variabilidade existente ao nível modelação das funcionalidades do sistema. No processo de implementação das mesmas funcionalidades, foram utilizados componentes totalmente reconfiguráveis, desenvolvidos em C++ *template metaprogramming*, para gerir a variabilidade existente ao nível do código das funcionalidades do sistema.

De seguida, são apresentados os passos seguidos no desenvolvimento da *framework* generativa de videovigilância:

i. Numa primeira fase, foram utilizadas técnicas de *Software Product Line* (SPL), para identificar as funcionalidades que são comuns a todas as configurações do sistema e as funcionalidades que são variáveis, e caracterizam cada uma das diferentes configurações do sistema. Para isso, foram utilizados diagramas de funcionalidades, uma vez que estes permitem modular as

funcionalidades do sistema, sem ter em consideração o tipo de mecanismo de gestão da variabilidade utilizado ao nível da implementação, tais como, herança, *templates* ou compilação condicional;

ii. De seguida, foram estudados os vários mecanismos que a linguagem de programação C++ utiliza, para gerir a variabilidade existente ao nível do código. Como a maioria dos sistemas de videovigilância são estáticos, isto é as funcionalidades do mesmo não variam em tempo de execução, optou-se por utilizar métodos de gestão da variabilidade do código, que são resolvidos no ato da compilação. Dos vários mecanismos estudados, chegou-se à conclusão de que as *templates* e o *template metaprogramming* é que apresentavam melhores resultados, no que diz respeito à gestão da variabilidade do código e ao desempenho do sistema.

iii. Uma vez decidido o mecanismo de gestão da variabilidade ao nível do código, foram desenvolvidos os componentes totalmente reconfiguráveis. Na implementação das funcionalidades comuns dos componentes, foram utilizadas funcionalidades básicas da linguagem de programação C++. Para implementar as funcionalidades variáveis dos componentes, foram utilizadas *templates* e C++ *template metaprogramming*. Ao utilizar *templates* e C++ *template metaprogramming* para gerir a variabilidade do código dos componentes, é compilado apenas o código necessário que implementa a funcionalidade pretendida do componente, fazendo com que o código do ficheiro executável final da aplicação alvo, seja o mais otimizado possível.

iv. Como a linguagem C++ *template metapogramming* (TMP) utiliza conceitos avançados da linguagem de programação C++ e a sua sintaxe é bastante complexa, foi desenvolvida a ferramenta de modelação TMPxML (*Template Markup Language*), que facilita o desenvolvimento de código TMP utilizando modelos. Os modelos são representados na linguagem TMPxML, que depois de validada pelo XML *schema* TMPxML, utiliza um transformador XSLT, que transforma o código TMPxML para código TMP.

iv. De seguida, foi desenvolvido o ambiente de desenvolvimento de sistemas de videovigilância inteligentes, que utiliza o perfil da aplicação para guiar no processo de desenvolvimento da mesma. Assim, ao utilizar o perfil da aplicação, no ato da compilação da *framework* generativa de videovigilância, é produzido um sistema totalmente otimizado para a aplicação alvo.

v. Por fim, foram realizados testes comparativos entre os métodos utilizados para a getão da variabilidade do código dos *middlewares* e das *frameworks* multimédia existentes e a *framework* generativa de videovigilância desenvolvida. Os testes, consistiram essencialmente em métricas de comparação ao nível da gestão e manutenção do código, e ao nível de métricas de desempenho do sistema. Para isso, foram realizados testes em máquinas com diferentes características de *hardware* e *software*, para diferentes configurações do sistema e em três versões de implementação diferentes: uma implementação em C, outra implementação em C++ e, por fim, uma implementação em TMP. Da análise dos resultados, conclui-se que a versão TMP apresenta resultados de desempenho muito semelhantes à versão em C otimizada (versão que utiliza compilação condicional *"if-def"* para remover as funcionalidades não utilizadas). Quanto à versão em C++, apresenta sempre piores resultados de desempenho que as versões em C e TMP. No que diz respeito à metrica de gestão de código, a versão TMP apresenta melhores resultados que as outras duas versões. Logo, não restam dúvidas que ao utlizar TMP para gerir a variabilidade da *framework* generativa, não é introduzido *overhead* no desempenho do sistema e o processo de manutenção do código torna-se numa tarefa mais simples e eficaz, do que os métodos utilizados nas versões C e C++.

Logo, conclui-se que a *framework* generativa de videovigilância apresentada nesta tese de doutoramento, constituí uma mais valia para o desenvolvimento de sistemas de videovigilância de terceira geração. Conclui-se também, que as técnicas de desenvolvimento de *Software Product Line* (SPL), *Model Driven Development* (MDD) e *Agile*, quando utilizadas conjuntamente no processo de desenvolvimento de sistemas complexos, apresentam muitas mais vantagens do que aquelas apresentadas, quando estas técnicas são utilizadas de forma isolada. Finalmente, conclui-se que o C++ *template metaprogramming* deixou de ser uma miragem no que concerne à gestão da variabilidade do código de sistemas complexos, passando a ser mesmo o método *"state of the art"* para a gestão da variabilidade do código em tempo de compilação. Prova disso, é o trabalho apresentado nesta tese de doutoramento e as novas funcionalidades disponibilizadas pelos principais compiladores de C++, Microsoft Visual C++ e GCC, nas áreas afins ao C++ *template metaprogramming*.

8.2 Trabalho Futuro

Como trabalho futuro, são propostas as seguintes linhas de investigação:

- Sendo ponto assente a adequação do C++ TMP na implementação deste tipo de sistemas para ambientes embebidos, propunha-se a exploração de arquiteturas usualmente utilizadas em ambientes de programação funcional, e posteriormente compará-las com a implementação descrita nesta tese de doutoramento. Nomeadamente, a implementação dos filtros como *monads/monad transformers* e a implementação do *pipeline* como uma pilha destes *monads/monad transformer*;

- Utilizar técnicas de desenvolvimento *craftsmanship software*, para promover o desenvolvimento de sistemas de videovigilância inteligentes distribuídos com um nível de complexidade mais elevado, com envolvimento de comunidades de vários *stakeholders*, onde de forma cooperativa são integradas eficientemente as várias perspetivas do sistema;

- Realização de testes mais exaustivos noutras plataformas com menos recursos e com um nível de configuração do sistema mais variado;

- Incorporar suporte de depuração à *framework* desenvolvida, visando não só obviar os problemas associados à depuração em ambiente C++ TMP, como também para melhor compreender em tempo-real os efeitos associados às alterações da figurabilidade do sistemas durante a modelação e depuração omnisciente, que permite compreender através de navegação *forward* e *backward*, o historial de alteração de configuração de um sistemas desenvolvido segundo um paradigma *Agile* SPL;

- Explorar a extensão da *framework* com a incorporação de estratégias dinâmicas de baixo custo, promovendo assim maior flexibilidade na modelação de aplicações de videovigilância inteligentes, que possam beneficiar das mesmas;

- Explorar a migração de alguns filtros para *hardware* em plataformas embebidas baseadas em FPGA e ao mesmo tempo potencializar/promover um modelo de programação unificado/transparente para a instanciação de filtros em *software* e em *hardware*.

Bibliografia

"Technical report on c++ performance." [Online]. Available: www2.research.att.com/~bs/performanceTR.pdf

D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004.

M. Acher, P. Lahire, S. Moisan, and J.-P. Rigault, "Tackling high variability in video surveillance systems through a model transformation approach," in *Modeling in Software Engineering, 2009. MISE '09. ICSE Workshop on*, may 2009, pp. 44 –49.

F. Afonso, "Operating System Fault Tolerance Support for Real-Time Embedded Applications," Ph.D. dissertation, University of Minho, Portugal, Jan. 2009.

I. Ahmad, Z. He, M. Liao, F. Pereira, and M.-T. Sun, "Special issue on video surveillance," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 18, no. 8, pp. 1001 –1005, aug. 2008.

V. Alves, G. Santos, D. Pires, A. C. Neto, F. Calheiros, J. Leal, S. Soares, V. Nepomuceno, and P. Borba, "From conditional compilation to aspects: A case study in software product lines migration."

AMPLE, "Ample: Aspect-oriented, model-driven, product line engineering." [Online]. Available: http://cordis.europa.eu/search/index.cfm?fuseaction=proj.document&PJ_RCN=8724342

R. Anisko, "Towards automatic partial evaluation for the c++ language," in *CSI Seminar*, 2002.

Axis, "Evolution of video surveillance systems." [Online]. Available: http://www.axis.com/products/video/about_networkvideo/evolution.htm

M. Barr, *Programming Embedded Systems in C and C++*. Sebastopol, CA, USA: O'Reilly Associates, 1998.

P. G. Bassett, "Frame-Based Software Engineering," no. civil, pp. 9–16, 1987.

——, *Framing Software Reuse: Lessons From the Real World*. Upper Saddle River, NJ, USA: Prentice-Hall, 1996.

D. Batory, R. Cardone, and Y. Smaragdakis, "Object-oriented frameworks and product-lines," 2000.

T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wasowski, "Feature-to-code mapping in two large product lines," in *SPLC*, 2010, pp. 498–499.

L. Bourdev, "Generic image library." [Online]. Available: http://www.boost.org/doc/libs/1_52_0/libs/gil/doc/index.html

L. Bourdev and J. Järvi, "Efficient run-time dispatching in generic programming with minimal code bloat," *Sci. Comput. Program.*, vol. 76, no. 4, pp. 243–257, Apr. 2011.

T. J. Brown, I. Spence, P. Kilpatrick, and D. Crookes, "Adaptable components for software product line engineering," in *Proceedings of the Second International Conference on Software Product Lines*, ser. SPLC 2. London, UK, UK: Springer-Verlag, 2002, pp. 154–175.

T. Buchmann and A. Dotor, "Mapping features to domain models in fujaba," in *Pieter van Gorp, Hrsg., Proceedings of the 7th International Fujaba Days*, Seiten 20-24, Eindhoven, The Netherlands, 2009.

N. Cardoso, N. Peixoto, J. Cabral, J. Mendes, J. Monteiro, and A. Tavares, "Aspect-oriented programming of video-based surveillance systems," in *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, july 2011, pp. 551 –556.

N. Cardoso, J. Cabral, P. Cardoso, J. Mendes, A. Tavares, and J. Monteiro, "A novel approach to manage the complexity and heterogeneity of video surveillance systems," in *Electrotechnical Conference (MELECON), 2012 16th IEEE Mediterranean*, march 2012, pp. 856 –861.

N. Cardoso, P. Garcia, T. Gomes, F. Salgado, P. Rodrigues, J. Cabral, J. Mendes, and A. Tavares, "Multi-camera home appliance network: Handling device interoperability," in *Industrial Informatics (INDIN), 2012 10th IEEE International Conference on*, july 2012, pp. 69 –74.

N. Cardoso, P. Rodrigues, O. Ribeiro, J. Cabral, J. Monteiro, M. J., and A. Tavares, "An agile software product line model-driven design environment for video surveillance systems," in *in the 17th IEEE International Conference on Emerging Technologies Factory Automation*, sep. 2012.

——, "Model-driven template metaprogramming," in *in the 17th IEEE International Conference on Emerging Technologies Factory Automation*, sep. 2012.

N. Cardoso, P. Rodrigues, J. Vale, M. Fernandes, P. Gonçalves, J. Mendes, and A. Tavares, "Framework especializada para sistemas de videovigilância orientada a sistemas embebidos," in *Seminario Anual de Automática, Electrónica Industrial e Instrumentación*, july 2012.

N. Cardoso, J. Vale, J. Cabral, J. Mendes, P. Cardoso, A. Tavares, and J. Monteiro, "Use of template metaprogramming to address the heterogeneity of video surveillance systems," in *Industrial Technology (ICIT), 2012 IEEE International Conference on*, march 2012, pp. 384 –389.

N. Cardoso, P. Rodrigues, J. Vale, P. Garcia, P. Cardoso, J. L. Monteiro, J. Cabral, J. Mendes, M. Ekpanyapong, and A. Tavares, "A generative-oriented model-driven design environment for customizable video surveillance systems." *EURASIP J. Emb. Sys.*, vol. 2012, p. 7, 2012.

W. Chao and X. M. Jun, "Multi-agent based distributed video surveillance system over ip," in *Computer Science and Computational Technology, 2008. ISCSCT '08. International Symposium on*, vol. 2, dec. 2008, pp. 97 –100.

P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley Professional, 2001.

A. Cockburn, *Agile software development*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," in *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, ser. CSMR '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 191–200.

K. Czarnecki and U. W. Eisenecker, "Components and generative programming (invited paper)," *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 2–19, Oct. 1999. [Online]. Available: http://doi.acm.org/10.1145/318774.318779

——, *Generative programming: methods, tools, and applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration through specialization and multi-level configuration of feature models," in *Software Process Improvement and Practice*, 2005, p. 2005.

X. Desurmont, J. Delaigle, A. Bastide, and B. Macq, "A generic flexible and robust approach for intelligent real-time video-surveillance systems," in *Proc. SPIE 5297, Real-Time Imaging VIII*, May 2004, pp. 134 – 141.

H. Detmold, A. Dick, K. Falkner, D. Munro, A. van den Hengel, and R. Morrison, "Scalable surveillance software architecture," in *Video and Signal Based Surveillance, 2006. AVSS '06. IEEE International Conference on*, nov. 2006, p. 103.

H. Detmold, A. Dick, K. Falkner, D. S. Munro, A. van den Hengel, and R. Morrison, "Middleware for video surveillance networks," in *Proceedings of the international workshop on Middleware for sensor networks*, ser. MidSens '06. New York, NY, USA: ACM, 2006, pp. 31–36.

K. Driesen and U. Hölzle, "The direct cost of virtual function calls in c++," *SIGPLAN Not.*, vol. 31, no. 10, pp. 306–323, Oct. 1996. [Online]. Available: http://doi.acm.org/10.1145/236338.236369

D. Duque, "Previsão e identificação de eventos de quebra de segurança em vídeo-vigilância," Ph.D. dissertation, University of Minho, Portugal, Jun. 2009.

eCos, "ecos homepage." [Online]. Available: http://ecos.sourceware.org/

J. M. Favre, "Understanding-in-the-large," in *Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, ser. WPC '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 29–.

G. Ferreira, F. Gaia, E. Figueiredo, and M. Maia, "On the use of feature-oriented programming for evolving software product lines: A comparative study," Sep. 2011.

E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas, "Evolving software product lines with aspects: an empirical study on design stability," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 261–270.

C. Gacek and M. Anastasopoules, "Implementing product line variabilities," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 3, pp. 109–117, May 2001.

M. Garcia-Valls, P. Basanta-Val, and I. Estevez-Ayres, "Adaptive real-time video transmission over dds," in *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, july 2010, pp. 130 –135.

D. D. Gennaro, *Advanced C++ Metaprogramming*. CreateSpace Independent Publishing Platform, 2011.

GeoVision, "Geovision homepage." [Online]. Available: http://www.geovision.com.tw/

J. Gray, "C ++ : Under the Hood C-like Structs C-like Structs with C ++ Features," 1994.

GStreamer, "Gstreamer homepage." [Online]. Available: http://gstreamer.freedesktop.org/

A. Gurtovoy and D. Abrahams, "The boost mpl library." [Online]. Available: http://www.boost.org/doc/libs/1_48_0/libs/mpl/doc/index.html

A. Hampapur, L. Brown, J. Connell, S. Pankanti, A. Senior, and Y. Tian, "Smart surveillance: Applications, technologies and implications," in *In IEEE Pacific-Rim Conference On Multimedia*, 2003.

D. Herity, "C++ in embedded systems: Myth and reality." [Online]. Available: http://www.eetindia.co.in/ARTICLES/1998FEB/PDF/EEIOL_1998FEB02_EMS_TA.pdf

Hex-Rays, "Ida: About." [Online]. Available: http://www.hex-rays.com/products/ida/index.shtml

F. Hunleth, "Building customizable middleware using aspect-oriented programming," 2002.

B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep., November 1990.

E.-A. Karlsson, Ed., *Software Reuse: A Holistic Approach*. New York, NY, USA: John Wiley Sons, 1995.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP*, 1997, pp. 220–242.

R. Klemm and G. Fettweis, "Bitstream processing for embedded systems using c++ metaprogramming," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, march 2010, pp. 909 –913.

C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, no. 2, pp. 131–183, 1992. [Online]. Available: http://doi.acm.org/10.1145/130844.130856

T. Lane, "libjpeg." [Online]. Available: http://libjpeg.sourceforge.net/

M. Lanoe and E. Senn, "Consumption analysis and estimation in the design of gstreamer based multimedia applications," in *Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on*, oct. 2012, pp. 1 –7.

W. C. Lim, "Effects of reuse on quality, productivity, and economics," *Software, IEEE*, vol. 11, no. 5, pp. 23–30, 1994.

C. Lin and Y. Tang, "Research and design of the intelligent surveillance system based on directshow and opencv," in *Consumer Electronics, Communications and Networks (CECNet), 2011 International Conference on*, april 2011, pp. 4307 –4310.

Linux, "Linux homepage." [Online]. Available: http://www.linux.org/

B. P. L. Lo, J. Sun, and S. A. Velastin, "Fusing visual and audio information in a distributed intelligent surveillance system for public transport systems," *Acta Automatica Sinica*, vol. 29, no. 3, pp. 393–407, 2003.

D. Lohmann, "Aspect-Awareness in the Development of Configurable System Software," Ph.D. dissertation, Friedrich-Alexander Universität Erlangen-Nürnberg, 2009.

D. Lohmann, W. Hofer, W. Schroder-Preikschat, J. Streicher, and O. Spinczyk, "Ciao: An aspect-oriented operating-system family for resource-constrained embedded systems," 2009.

R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski, "Evolution of the linux kernel variability model," in *Proceedings of the 14th international conference on Software product lines: going beyond*, ser. SPLC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 136–150.

S. Meyers, "Effective c++ in an embedded environment." [Online]. Available: http://www.aristeia.com/c++-in-embedded.html

——, *Effective STL: 50 specific ways to improve your use of the standard template library*. Essex, UK, UK: Addison-Wesley Longman Ltd., 2001.

D. Muthig and T. Patzke, "Generic implementation of product line components," in *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, ser. NODe '02. London, UK, UK: Springer-Verlag, 2003, pp. 313–329.

P. Nuno, C. Nuno, C. Jorge, T. Adriano, and M. Jose, "A segmentation approach for object detection on highly dynamic aquatic environments," in *Industrial Electronics, 2009. IECON '09. 35th Annual Conference of IEEE*, nov. 2009, pp. 1985 –1989.

NUUO, "Nuuo homepage." [Online]. Available: http://www.nuuo.com/

D. Ostheimer, S. Lemay, M. Ghazal, D. Mayisela, A. Amer, and P. F. Dagba, "A modular distributed video surveillance system over ip," in *Electrical and Computer Engineering, 2006. CCECE '06. Canadian Conference on*, may 2006, pp. 518 –521.

M. A. Patricio, J. Carbó, O. Pérez, J. García, and J. M. Molina, "Multi-agent framework in visual sensor networks," *EURASIP J. Appl. Signal Process.*, vol. 2007, no. 1, pp. 226–226, Jan. 2007.

S. J. Paul, P. Bassett, H. Zhang, and W. Zhang, "Xvcl: Xml-based variant configuration language," in *Proc. Int. Conf. on Software Engineering, ICSEâ^{TM}03*, 2003, pp. 810–811.

N. Peixoto, N. Cardoso, P. Goncalves, P. Cardoso, J. Cabral, A. Tavares, and J. Mendes, "Motion segmentation object detection in complex aquatic scenes

and its surroundings," in *Industrial Informatics (INDIN), 2012 10th IEEE International Conference on*, july 2012, pp. 162 –166.

N. Peixoto, "Videovigilância Inteligente em Ambientes Aquáticos: Detecção Precoce de Afogamento em Piscinas Domésticas," Ph.D. dissertation, University of Minho, Portugal, 2010.

W. B. Pennebaker and J. L. Mitchell, *JPEG Still Image Data Compression Standard*, 1st ed. Norwell, MA, USA: Kluwer Academic Publishers, 1992.

M. D. Pesce, *Programming Microsoft DirectShow for Digital Video, Television, and DVD*. Redmond, WA, USA: Microsoft Press, 2002.

P. L. Plauger, "Embedded c++: An overview," *Embedded Systems Programming*, pp. 10–13, 1997.

K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, 2005.

T. Raty, "Survey on contemporary remote surveillance systems for public safety," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 40, no. 5, pp. 493 –515, sept. 2010.

D. Saks, "Reducing run-time overhead in c++ programs," in *Embedded Systems Conference*, San Francisco, 2002.

V. T. Sarinho and A. L. Apolinário, "A generative programming approach for game development," in *Proceedings of the 2009 VIII Brazilian Symposium on Games and Digital Entertainment*, ser. SBGAMES '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 83–92.

scitools, "Understand source code analysis metrics." [Online]. Available: http://www.scitools.com/index.php

M. Sedky, M. Moniri, and C. Chibelushi, "Classification of smart video surveillance systems for commercial applications," in *Advanced Video and Signal Based Surveillance, 2005. AVSS 2005. IEEE Conference on*, sept. 2005, pp. 638 – 643.

T. Senst, M. Patzold, R. Evangelio, V. Eiselein, I. Keller, and T. Sikora, "On building decentralized wide-area surveillance networks based on onvif," in *Advanced Video and Signal-Based Surveillance (AVSS), 2011 8th IEEE International Conference on*, 30 2011-sept. 2 2011, pp. 420 –423.

S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "The variability model of the linux kernel," in *VaMoS*, 2010, pp. 45–51.

SimuNova, "Matrix template library." [Online]. Available: http://www.simunova.com/en/node/24

T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

C. Steup, M. Schulze, and J. Kaiser, "Exploiting Template-Metaprogramming for Highly Adaptable Device Drivers a Case Study on CANARY an AVR CAN-Driver," in *12th Brazilian Workshop on Real-Time and Embedded Systems (WTR)*. Gramado, Brazil: Brazilian Computer Society, May 24 2010.

M. Valera and S. Velastin, "Intelligent distributed surveillance systems: a review," *Vision, Image and Signal Processing, IEE Proceedings* -, vol. 152, no. 2, pp. 192 – 204, april 2005.

S. Velastin, L. Khoudour, B. Lo, J. Sun, and M. Vicencio-Silva, "Prismatica: a multi-sensor surveillance system for public transport networks," in *Road Transport Information and Control, 2004. RTIC 2004. 12th IEE International Conference on*, april 2004, pp. 19 – 25.

S. Velastin, B. Boghossian, B. Lo, J. Sun, and M. Vicencio-Silva, "Prismatica: toward ambient intelligence in public transport environments," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 35, no. 1, pp. 164 – 182, jan. 2005.

H. Wada, E. M. M. Babu, A. Malinowski, J. Suzuki, and K. Oba, "Design and implementation of the matilda distributed uml virtual machine," in *IASTED Int'l Conf. on Software Engineering and Applications*, 2006.

G. Wang, L. Tao, H. Di, X. Ye, and Y. Shi, "A scalable distributed architecture for intelligent vision system," *Industrial Informatics, IEEE Transactions on*, vol. 8, no. 1, pp. 91 –99, feb. 2012.

M. Wang and M. Parashar, "Object-oriented stream programming using aspects," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010, pp. 1 –11.

Y.-K. Wang, C.-T. Fan, and C.-R. Huang, "A large scale video surveillance system with heterogeneous information fusion and visualization for wide area monitoring," in *Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP), 2012 Eighth International Conference on*, july 2012, pp. 178 –181.

D. M. Weiss and C. T. R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

R. Wijnhoven, E. Jaspers, and P. de With, "Multi-channel video streaming server for surveillance systems," in *Consumer Electronics, 2004 IEEE International Symposium on*, 1-3, 2004, pp. 353 – 358.

J. Withey, "Investment Analysis of Software Assets for Product Lines Software Engineering Institute," no. November, 1996.

XVCL, "Xml-based variant configuration language homepage." [Online]. Available: http://xvcl.comp.nus.edu.sg/cms/

L. Yuan, J. S. Dong, and J. Sun, "Modeling and customization of fault tolerant architecture using object-z/xvcl," in *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*, dec. 2006, pp. 209 –216.

H. Zhang and S. Jarzabek, "A hybrid approach to feature-oriented programming in xvcl," in *Proceedings of the 14th international conference on Software product lines: going beyond*, ser. SPLC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 440–445.

F. Ziliani, "The importance of 'scalability' in video surveillance architectures," in *Imaging for Crime Detection and Prevention, 2005. ICDP 2005. The IEE International Symposium on*, june 2005, pp. 29 – 32.

Apêndice A

Implementação de um Filtro

Os filtros são os componentes elementares e os pontos de extensibilidade da *framework* generativa de videovigilância. Sempre que seja necessário adicionar uma nova funcionalidade não suportada pela *framework*, a forma mais simples e segura de o fazer, é desenvolver um novo filtro que implemente a funcionalidade pretendida.

Para descrever os passos necessários no processo de desenvolvimento de um novo filtro, é dado como exemplo a implementação do filtro de entrada V4L2. O filtro de entrada V4L2, permite fazer a captura de imagens de câmaras analógicas em formato descomprimido, no sistema operativo Linux.

A.1 Diagrama de Classes

Tal como ilustra o diagrama de classes da Figura A.1, a classe *template* CV4L2Source, que representa o filtro de entrada V4L2, herda da classe *template* CBaseFilter. A classe *template* CBaseFilter implementa as funcionalidades básicas de um filtro e recebe dois parâmetros de entrada de *template*: o tipo de filtro e um vetor com os tipos de dados de cada um dos pinos que fazem parte do filtro. Neste caso específico, o tipo de filtro é definido como um filtro de entrada (source_filter_t) e o vetor de pinos é constituído apenas por um pino do tipo CV4l2OutputPin.

A classe *template* CV4L2Source recebe dois parâmetros de entrada de *template*: o tipo de dados IOMethod e o tipo de dados FormatType. O tipo de dados IOMethod define o método de captura utilizado pelo filtro de entrada V4L2. O tipo de dados

FormatType define o formato de imagem utilizado na captura de vídeo.

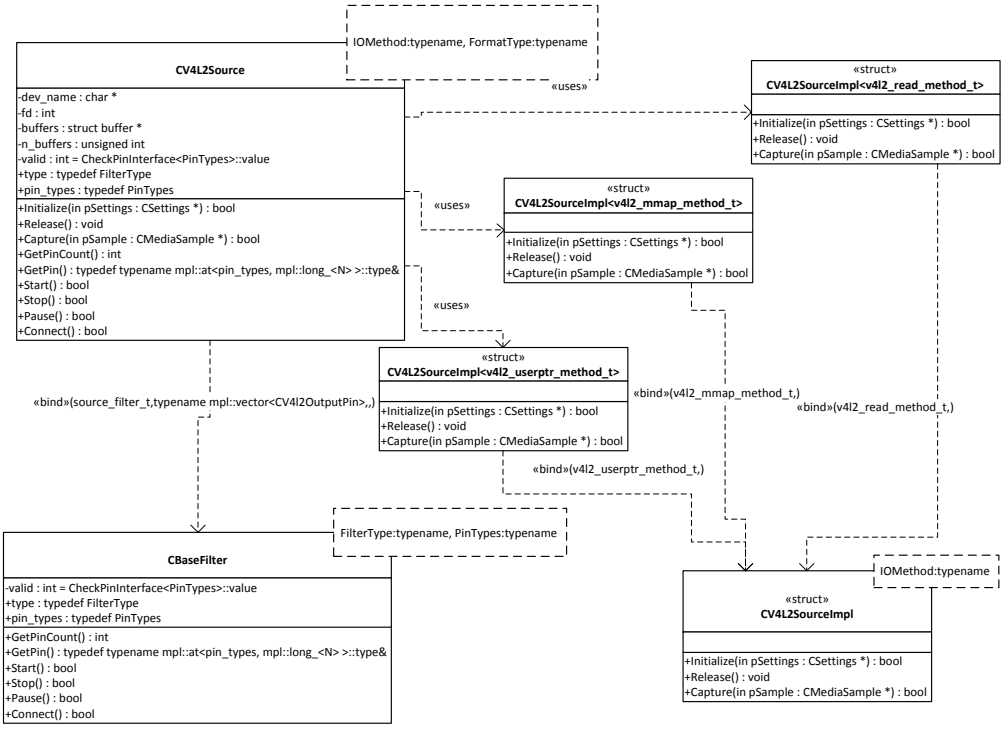


Figura A.1: Diagrama de classes do filtro de entrada V4L2.

Para gerir a variabilidade dos diferentes métodos de captura do filtro de entrada V4L2, foi definida a estrutura *template* CV4L2SourceImpl. A estrutura *template* CV4L2SourceImpl implementa três *templates* especializadas, uma para cada um dos métodos de captura disponíveis. No ato de compilação, é instanciada a *template* especializada da estrutura *template* CV4L2SourceImpl com o tipo de dados IOMethod da classe *template* CV4L2Source.

O código esqueleto da classe *template* CV4L2Source, é apresentado nas seguintes linhas de código:

```
template<typename IOMethod, typename FormatType>
class CV4L2Source
    : public CBaseFilter<source_filter_t, typename mpl::vector<CV4l2OutputPin> >
{
public:
    CV4L2Source();
    ~CV4L2Source();

    bool Initialize(CSettings *pSettings) {
        ...
        fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        fmt.fmt.pix.width = pSettings->image_width;
        fmt.fmt.pix.height = pSettings->image_height;
        fmt.fmt.pix.pixelformat = FormatType::value;
```

```
        ...
        return CV4L2SourceImpl<IOMethod>::Initialize<CV4L2Source>(*this);
    }

    void Release() {
        CV4L2SourceImpl<IOMethod>::Release<CV4L2Source>(*this);
    }

    bool Capture(CMediaSample *pSample) {
        return CV4L2SourceImpl<IOMethod>::Capture<CV4L2Source>(*this, pSample);
    }
    ...
};
```

Não tomando em consideração o método construtor e o método destrutor, a classe *template* CV4L2Source implementa três métodos: o método Initialize, o método Release e o método Capture. O método Initialize inicializa o filtro de entrada V4L2 em função do método utilizado para fazer a captura dos dados. Para isso, usa o método *template* Initialize da estrutura *template* CV4L2SourceImpl. O método Release liberta os recursos alocados pelo filtro de entrada V4L2, usando para isso o método Release da estrutura *template* CV4L2SourceImpl. Finalmente, o método Capture faz a captura de uma imagem usando o método Capture da estrutura *template* CV4L2SourceImpl. O método Capture da estrutura *template* CV4L2SourceImpl, recebe como parâmetro de entrada uma referência do filtro de entrada e um apontador para a classe CMediaSample que contém os dados da imagem adquirida.

A.2 Métodos de Captura

O parâmetro de entrada de *template* IOMethod da classe *template* CV4L2Source, define o método de captura utilizado pelo filtro de entrada V4L2. O driver V4L2 define três métodos de captura: o método read, o método mmap e o método userptr. As estruturas que definem cada um dos métodos de captura, são apresentadas nas linhas de código seguintes:

```
// define o método de captura read
struct v4l2_read_method_t {};

// define o método de captura mmap
struct v4l2_mmap_method_t {};

// define o método de captura userptr
struct v4l2_userptr_method_t {};
```

A.3 Formatos de Captura

O parâmetro de entrada de *template* FormatType da classe *template* CV4L2Source, define o formato de captura utilizado pelo filtro de entrada V4L2. Como a placa de captura utilizada suporta apenas três formatos de captura (YUYV, GREY e YUV420), foram definas as respetivas estruturas que representam cada um dos formatos de captura:

```
// define o formato de captura YUYV
struct v4l2_yuyv_format_t {
    value = V4L2_PIX_FMT_YUYV;
};

// define o formato de captura Grey
struct v4l2_grey_format_t {
    value = V4L2_PIX_FMT_GREY;
};

// define o formato de captura YUV420
struct v4l2_yuv420_format_t {
    value = V4L2_PIX_FMT_YVU420;
};
```

A.4 Estrutura *Template CV4L2SourceImpl*

A estrutura *template* CV4L2SourceImpl é utilizada pela classe *template* CV4L2Source para implementar cada um dos métodos da classe, em função do método de captura definido pelo parâmetro de entrada de *template* IOMethod. Assim, são definidas três *templates* especializadas que implementam os métodos Initialize, Release e Capture, para cada um dos métodos de captura definidos.

Para implementar a *template* especializada da classe *template* CV4L2SourceImpl, que utiliza o método de captura read, são necessárias as seguintes linhas de código:

```
template<>
struct CV4L2SourceImpl<v4l2_read_method_t> {
    template<typename V4L2Source>
    static inline int Initialize(V4L2Source& v4l2) {
        // TODO: code to initialize V4L2 source filter using read method
    }

    template<typename V4L2Source>
    static inline int Capture(V4L2Source& v4l2, CMediaSample *pSample) {
        // TODO: code to capture image from V4L2 source filter using read method
    }
```

```
    template<typename V4L2Source>
    static inline int Release(V4L2Source& v4l2) {
        // TODO: code to finalize V4L2 source filter using read method
    }
};
```

Para implementar a *template* especializada da classe *template* CV4L2SourceImpl, que utiliza o método de captura mmap, são necessárias as seguintes linhas de código:

```
template<>
struct CV4L2SourceImpl<v4l2_mmap_method_t> {
    template<typename V4L2Source>
    static inline int Initialize(V4L2Source& v4l2) {
        // TODO: code to initialize V4L2 source filter using mmap method
    }

    template<typename V4L2Source>
    static inline int Capture(V4L2Source& v4l2, CMediaSample *pSample) {
        // TODO: code to capture image from V4L2 source filter using mmap method
    }

    template<typename V4L2Source>
    static inline int Release(V4L2Source& v4l2) {
        // TODO: code to finalize V4L2 source filter using mmap method
    }
};
```

Finalmente, o código necessário para implementar a *template* especializada da classe *template* CV4L2SourceImpl, que utiliza o método de captura userptr, é apesentado de seguida:

```
template<>
struct CV4L2SourceImpl<v4l2_userptr_method_t> {
    template<typename V4L2Source>
    static inline int Initialize(V4L2Source& v4l2) {
        // TODO: code to initialize V4L2 source filter using userptr method
    }

    template<typename V4L2Source>
    static inline int Capture(V4L2Source& v4l2, CMediaSample *pSample) {
        // TODO: code to capture image from V4L2 source filter using userptr method
    }

    template<typename V4L2Source>
    static inline int Release(V4L2Source& v4l2) {
        // TODO: code to finalize V4L2 source filter using userptr method
    }
};
```

A.5 Implementação do Pino de Saída

O filtro de entrada V4L2 é constituído apenas por um pino de saída, o pino CV4L2OutputPin. Tal como foi referido anteriormente, o pino é a entidade responsável pelas ligações entre os filtros. Assim, sempre que for necessário estabelecer uma ligação entre o filtro de entrada CV4L2Source e um outro filtro do mesmo *filter graph*, é o pino CV4L2OutputPin que toma a iniciativa de estabelecer a ligação (pino ativo). Como o pino CV4L2OutputPin é um pino de saída e pertence a um filtro de entrada, é responsável pela criação de uma *thread* que faz a captura de imagens, e de seguida, envia as imagens para os restantes filtros do *filter graph*.

As linhas de código seguintes definem a classe CV4L2OutputPin, que implementa o pino de saída do filtro de entrada V4L2:

```
class CV4L2OutputPin
    : public CBasePin<output_pin_t,
        typename mpl::vector<YUYV_MEDIA_TYPE, GREY_MEDIA_TYPE, YUV420_MEDIA_TYPE> >
{
public:
    CV4l2OutputPin() {
        setThreadProc(this, ThreadProc);
    };
    ~CV4l2OutputPin() { ... }

    static void ThreadProc(void *args) {
        CV4l2OutputPin *pThis = (CV4l2OutputPin *)args;
        pThis->Run();
    }
    ...
};
```

Tal como todos os pinos implementados na *framework* generativa de videovigilância, a classe CV4L2OutputPin herda da classe *template* CBasePin. A classe *template* CBasePin implementa as funcionalidades básicas de um pino e recebe como parâmetros de entrada de *template*: o tipo de dados PinType e o vetor MediaTypes. O tipo de dados PinType define o tipo de pino. O vetor MediaTypes define os tipos de *media* de áudio/vídeo suportados pelo pino. Neste caso específico, como é um pino de saída, o tipo de dados PinType é definido como output_pin_t e o vetor MediaTypes é definido com os tipos de *media* de vídeo YUYV_MEDIA_TYPE, GREY_MEDIA_TYPE e YUV420_MEDIA_TYPE.

Como o pino CV4L2OutputPin é do tipo pino de saída e pertence a um filtro de entrada, herda da classe CThread, que permite criar uma nova *thread* de processamento para a aquisição de dados. No construtor da classe *template* é definido o

método estático ThreadProc como a função principal de processamento da *thread*, que executa o método Run da classe *template* CV4L2OutputPin.

```
void CV4l2OutputPin::Run() {
   CMediaSample *pSample = new CMediaSample();

   while(1) {
     ...
     pFilter->Capture(pSample);
     if (m_pPin != NULL)
                m_pFuncProcess(m_pPin, pSample);
     ...
   }
}
```

Numa primeira fase, o método Run da classe *template* CV4L2OutputPin cria o objeto pSample que contém informações sobre a imagem adquirida. De seguida, é executado um ciclo infinito para fazer a captura das imagens através do método Capture da classe *template* CV4L2Source. Caso o pino esteja ligado a um outro pino, é enviado ao pino um apontador do objeto pSample com as informações da imagem adquirida.

As linha de código seguinte, criam um filtro de entrada do tipo V4L2:

```
CV4L2Source<v4l2_mmap_method_t, v4l2_yuv420_format_t> sourceFilter;
```

O parâmetro de entrada de *template* v4l2_mmap_method_t define como método de captura, o método mmap. O parâmetro de entrada de *template* v4l2_yuv420_format_t define como formato de captura, o formato YUV420.

Apêndice B

Schema XML da linguagem TMPxML

O *Schema* XML da linguagem TMPxML, define um conjunto de regras para validação de documentos escritos na linguagem TMPxML. As regras de validação do documento são definidas por um conjunto de elementos, que representam os blocos básicos da linguagem TMP, apresentados no capítulo 5.

A tabela B.1, apresenta uma breve descrição de cada um dos elementos definidos no *Schema* XML da linguagem TMPxML:

Tabela B.1: Elementos definidos no *Schema* XML da linguagem TMPxML.

Elemento	**Descrição**
Module	define a raiz de qualquer documento TMPxML
Template	define o nível de topo das estruturas ou classes *template*
Struct	define uma estrutura *template*
Class	define uma classe *template*
Parameter	define um parâmetro de entrada de uma *template*
Extension	define uma estrutura ou uma classe *template* base
Code	define o bloco de código da *template*
Specialization	define uma *template* totalmente ou parcialmente especializada
Typedef	define uma variável TMP do tipo *typedef*
Variable	define uma variável TMP do tipo inteiro

De seguida, é feita uma descrição mais pormenorizada de cada um dos elementos apresentados na tabela B.1.

B.1 Elemento Module

O elemento module representa o elemento pai (raiz) de todos os elementos de um documento TMPxML.

A Figura B.1 apresenta o *schema* XSD do elemento module:

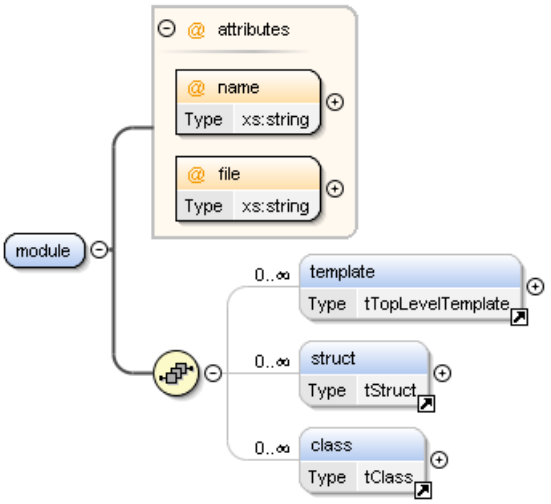


Figura B.1: *Schema* XSD do elemento *module*.

Tal como ilustra a figura B.1, o elemento module é constituído por dois atributos e três elementos. O atributo name define o nome do módulo (campo obrigatório). O atributo file define o ficheiro de saída onde o código TMP é gravado (campo obrigatório). O elemento template permite criar *inner templates* (campo opcional). O elemento struct cria uma estrutura *template* (campo opcional). Por fim, o elemento class cria uma classe *template* (campo opcional).

O código seguinte apresenta um exemplo da definição em XML do elemento module:

```
<module name="FilterTypes" file="filter_types.hpp">
   ...
</module>
```

O código apresentado define o módulo com o nome FilterTypes, utilizando a *tag* XML name. A *tag* XML file, especifica que o resultado da transformação do código do modulo, isto é da linguagem TMPxML para a linguagem TMP, é escrito no ficheiro filter_types.hpp.

B.2 Elemento Template

O elemento template representa o elemento de nível de topo, que define estruturas ou classes *template*.

A Figura B.2 apresenta o *schema* XSD do elemento template:

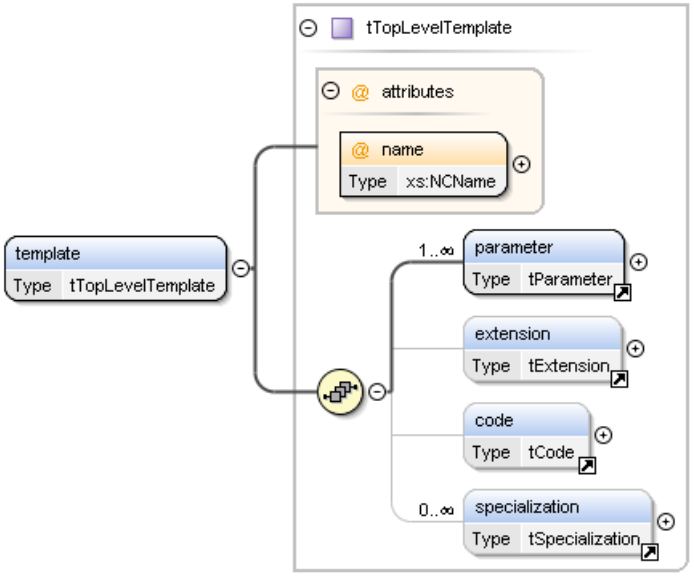


Figura B.2: *Schema* XSD do elemento *template*.

No diagrama da figura B.2, o elemento template é definido como um tipo complexo, com o nome tTopLevelTemplate. O tipo complexo tTopLevelTemplate é constituído por um atributo e quatro elementos. O atributo name define o nome da *template* (campo obrigatório). O elemento parameter define os parâmetros de entrada da *template* (campo opcional). O elemento extension permite estender uma nova *template* de uma outra já existente, através da herança (campo opcional). O elemento code define o bloco de código da *template* (campo obrigatório). Por fim, o elemento specialization permite criar *templates* totalmente ou parcialmente especializadas (campo opcional).

O código seguinte apresenta um exemplo da definição em XML do elemento template:

```
<template name="source_filter_t">
   ...
</template>
```

O código apresentado define a *template* com o nome source_filter_t, utilizando a *tag* XML name.

B.3 Elemento Struct e o Elemento Class

Em linguagem de programação C++, o único fator diferenciador entre uma estrutura e uma classe, é que numa estrutura, as variáveis e métodos são definidos por defeito como públicos e numa classe, são definidos por defeito como privados. Assim, os elementos struct e class partilham o mesmo bloco de definição em TMPxML.

A Figura B.3 apresenta o *schema* XSD do elemento struct:

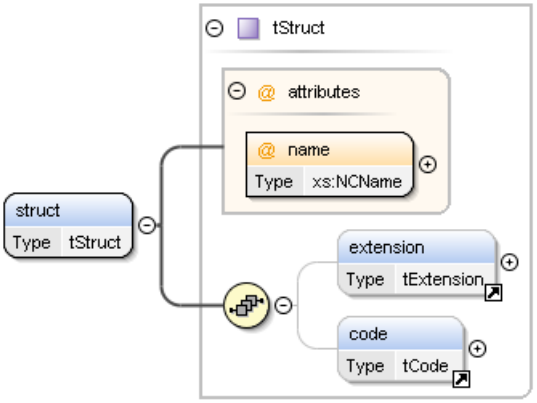


Figura B.3: *Schema* XSD do elemento *struct*.

No diagrama da figura B.3, o elemento struct é definido como um tipo complexo, com o nome tStruct. O tipo complexo tStruct é constituído por um atributo e dois elementos. O atributo name define o nome da estrutura (campo obrigatório). O elemento code define o código do corpo da estrutura (campo obrigatório). Por fim, o elemento extension permite que a estrutura herde de uma estrutura já existente (campo opcional).

O código seguinte apresenta um exemplo da definição em XML do elemento struct:

```
<struct name="link_filters">
   ...
</struct>
```

O código apresentado define a estrutura de dados com o nome link_filters, utilizando a *tag* XML name. A definição de uma classe segue o mesmo padrão que a definição de uma estrutura, mas em vez de se utilizar a *tag* XML struct, utiliza-se a *tag* XML class.

B.4 Elemento Parameter

O elemento parameter define um parâmetro de entrada de uma *template*. Assim, se uma *template* é definida com três parâmetros de entrada, é necessário criar três elementos deste tipo, um para cada parâmetro de entrada.

A Figura B.4 apresenta o *schema* XSD do elemento parameter:

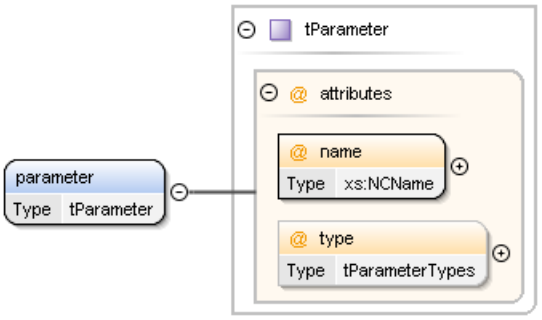


Figura B.4: *Schema* XSD do elemento *parameter*.

No diagrama da figura B.4, o elemento parameter é definido como um tipo complexo, com o nome tParameter. O tipo complexo tParameter é constituído por dois atributos. O atributo name define o nome do parâmetro de entrada (campo obrigatório). Por fim, o atributo type define o tipo de dados do parâmetro de entrada (campo obrigatório).

O código seguinte apresenta um exemplo da definição em XML do elemento parameter:

```
<parameter name="number_of_pins" type="int"/>
```

O código apresentado define um parâmetro de entrada de *template*. A *tag* XML name especifica que o nome do parâmetro de entrada é number_of_pins. A *tag* XML type especifica que o tipo de dados do parâmetro de entrada é um int.

B.5 Elemento Extension

O elemento extension define a estrutura ou classe *template* base, de onde uma *template* herda.

A Figura B.5 apresenta o *schema* XSD do elemento extension:

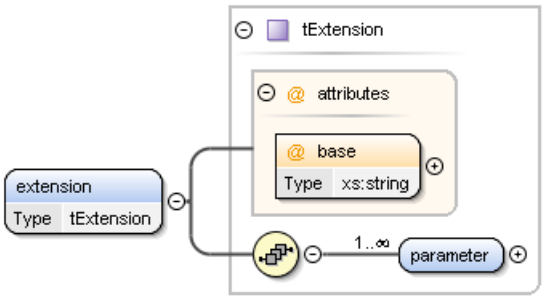


Figura B.5: *Schema* XSD do elemento *extension*.

No diagrama da figura B.5, o elemento extension é definido como um tipo complexo, com o nome tExtension. O tipo complexo tExtension é constituído por um atributo e por um ou mais elementos do tipo parameter. O atributo base define a estrutura ou classe *template* base (campo obrigatório). Por fim, o(s) elemento(s) parameter define(m) o(s) parâmetro(s) de entrada da estrutura ou classe *template* base (campo obrigatório).

O código seguinte apresenta um exemplo da definição em XML do elemento extension:

```
<extension base="source_filter_t"/>
```

O código apresentado define que uma determinada estrutura ou classe *template* herda da estrutura *template* source_filter_t. Como a estrutura *template* base não tem parâmetros de entrada, não é apresentada nenhuma *tag* XML parameter filha da *tag* XML extension.

B.6 Elemento Code

O elemento code define o bloco de código da *template*. A Figura B.6 apresenta o *schema* XSD do elemento code:

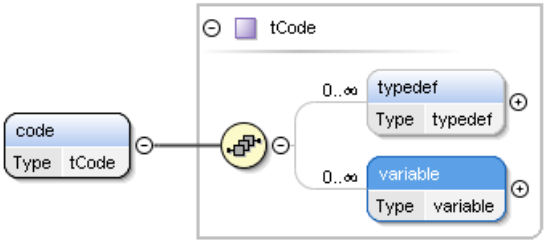


Figura B.6: *Schema* XSD do elemento *code*.

No diagrama da figura B.6, o elemento code é definido como um tipo complexo, com o nome tCode. O tipo complexo tCode é constituído por dois elementos, o elemento typedef e o elemento variable, que são descritos mais à frente.

O código seguinte apresenta um exemplo da definição em XML do elemento code:

```
<code>
   ...
</code>
```

O código apresentado define que uma determinada estrutura ou classe *template* contém o código definido dentro da *tag* XML code.

B.7 Elemento Specialization

O elemento specialization define uma *template* totalmente ou parcialmente especializada. Este elemento é opcional porque nem todas as *templates* são especializadas.

A Figura B.7 apresenta o *schema* XSD do elemento specialization:

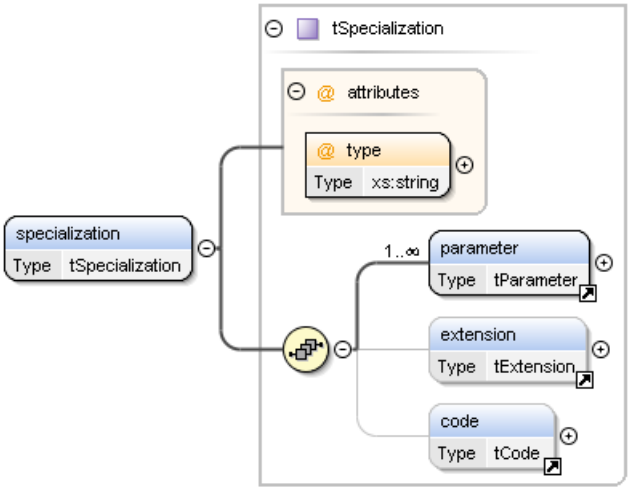


Figura B.7: *Schema* XSD do elemento *specialization*.

No diagrama da figura B.7, o elemento specialization é definido como um tipo complexo, com o nome tSpecialization. O tipo complexo tSpecialization é constituído por um atributo e três elementos. O atributo type define o tipo de especialização (campo obrigatório). O elemento parameter define os argumentos de entrada. O elemento extension define a estrutura ou classe *template* base (campo opcional). Finalmente, o elemento code implementa o código da *template*.

O código seguinte apresenta um exemplo da definição em XML do elemento specialization:

```
<specialization type="T,T">
   ...
</specialization>
```

O código apresentado, define que uma determinada estrutura ou classe *template*, contém uma *template* especializada do tipo <T,T>.

B.8 Elemento Typedef

O elemento typedef define a variáveis TMP do tipo *typedef*.

A Figura B.8 apresenta o *schema* XSD do elemento typedef:

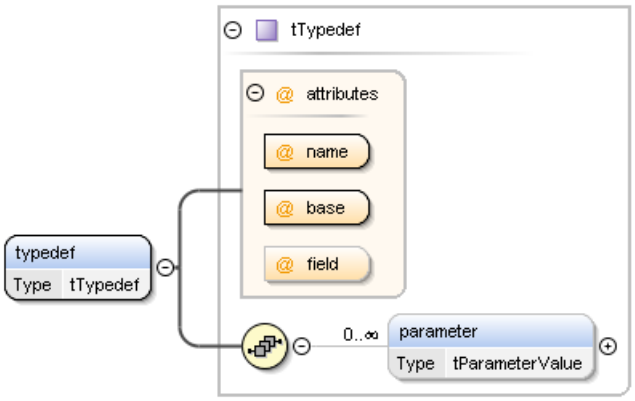


Figura B.8: *Schema* XSD do elemento *typedef*.

No diagrama da figura B.8, o elemento typedef é definido como um tipo complexo, com o nome tTypedef. O tipo complexo tTypedef é constituído por três atributos e elementos opcionais do tipo parameter. O atributo name define o novo tipo de dados (campo obrigatório). O atributo base define o tipo base (campo obrigatório). O atributo field define o subtipo do tipo base (campo obrigatório). O elemento parameter permite passar parâmetros de entrada para uma metafunção definida como o tipo base (campo opcional).

O código seguinte apresenta um exemplo da definição em XML do elemento typedef:

```
<typedef name="value" base="int"/>
```

O código apresentado define a variáveis TMP do tipo *typedef* com o nome value com o tipo de dados int.

B.9 Elemento Variable

O elemento variable define a variáveis TMP do tipo inteiro.

A Figura B.9 apresenta o *schema* XSD do elemento variable:

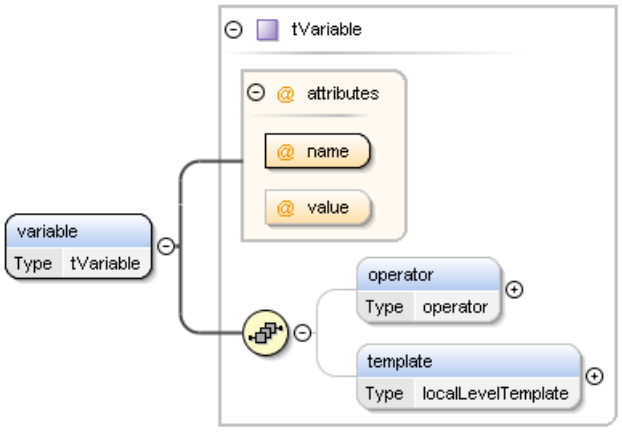


Figura B.9: *Schema* XSD do elemento *variable*.

No diagrama da figura B.9, o elemento variable é definido como um tipo complexo, com o nome tVariable. O tipo complexo tVariable é constituído por dois atributos e dois elementos. O atributo name define o nome da variável (campo obrigatório). O atributo value define o valor da variável (campo obrigatório). O elemento operator define o operador a utilizar (campo opcional). Por fim, o elemento tenplate faz referência a valores definidos em determinadas operações com *templates* (campo opcional).

O código seguinte apresenta um exemplo da definição em XML do elemento variable:

```
<variable name="value" value="2"/>
```

O código apresentado define a variáveis TMP do tipo *variable* com o nome value e o valor 2.

B.10 Função Factorial

Para perceber melhor cada um dos elementos apresentados anteriormente e a forma como estes devem ser definidos, é apresentado o código TMPxML da função factorial:

```
<template name="factorial">
   <parameter name="N" type="int"/>
   <code>
      <variable name="value">
         <operator type="*">
            <variable value="N"/>
            <recursion value="N?1"/>
         </operator>
      </variable>
   </code>
   <specialization type="0">
      <code>
         <variable name="value" value="1"/>
      </code>
   </specialization>
</template>
```

No código apresentado, é definida a estrutura *template* com nome factorial. A estrutura template factorial, recebe como parâmetro de entrada a variável com o nome N do tipo inteiro, definida na *tag* XML parameter. O código da *template*, é definido dentro da *tag* XML code, onde é definida uma variável com o nome value, que toma o valor do resultado da operação *. A operação * recebe dois parâmetros de entrada: a variável N e o resultado da função recursiva N-1.

Sabendo que a função factorial é recursiva, é definida a condição de paragem da estrutura *template* factorial utilizando a *tag* XML specialization. Neste caso específico, a condição de paragem acontece quando o valor do parâmetro de entrada de *template* N é 0. O valor de retorno da metafunção é 1, o valor definido na variável value, através da *tag* XML variable.