# DISSERTAÇÃO DE MESTRADO


**UNIVERSIDADE FEDERAL DE SÃO CARLOS**

**CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA**

**PROGRAMA DE PÓS-GRADUAÇÃO EM**

**CIÊNCIA DA COMPUTAÇÃO**


## “Refatoração com Enfoque em Portabilidade como Estratégia de Reúso em Desenvolvimento de Middleware para TV Digital Interativa”


**ALUNO:** Eduardo Drumond Sardinha
**ORIENTADOR:** Prof. Dr. César A. C. Teixeira



**São Carlos**
**Dezembro/2011**

**SÃO CARLOS - SP**
**BRASIL**

**UNIVERSIDADE FEDERAL DE SÃO CARLOS**

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

# REFATORAÇÃO COM ENFOQUE EM PORTABILIDADE COMO ESTRATÉGIA DE REÚSO EM DESENVOLVIMENTO DE MIDDLEWARE PARA TV DIGITAL INTERATIVA

**EDUARDO DRUMOND SARDINHA**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos como parte dos requisitos para a obtenção de título de Mestre em Ciência da Computação, área de concentração: Sistemas Distribuídos e Redes

Orientador: Prof. Dr. César A. C. Teixeira

São Carlos - SP

Dezembro/2011

**Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar**

S244re

Sardinha, Eduardo Drumond.
Refatoração com enfoque em portabilidade como estratégia de reúso em desenvolvimento de middleware para TV digital interativa / Eduardo Drumond Sardinha. -- São Carlos : UFSCar, 2012.
69 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2011.

1. Software. 2. Reuso. 3. Televisão digital. 4. Middleware. 5. Refatoração. I. Título.

CDD: 005.3 (20ª)

# Universidade Federal de São Carlos

## Centro de Ciências Exatas e de Tecnologia

### Programa de Pós-Graduação em Ciência da Computação

# "Refatoração com enfoque em Portabilidade como Estratégia de Reuso em Desenvolvimento de Middleware para TV Digital Interativa"

EDUARDO DRUMOND SARDINHA

**Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação**

**Membros da Banca:**

**Prof. Dr. Cesar Augusto Camillo Teixeira**
**( Orientador - DC/UFSCar)**

**Prof. Dr. Antonio Francisco do Prado**
**(DC/UFSCar)**

**Prof. Dr. Marco Antônio Pinheiro de Cristo**
**(UFAM)**

**São Carlos**
**Dezembro/2011**

Dedicado aos meus pais e Paula, com muito amor.

# AGRADECIMENTOS

A Deus, por tudo; à minha família; ao querido amigo/irmão Andreson Reis; à Fundação Centro de Análise, Pesquisa e Inovação (FUCAPI), pela oportunidade de fazer o Mestrado e pelos meses que precisei ficar ausente da instituição em Manaus em função dos estudos; aos colegas do Departamento de Tecnologia da FUCAPI que contribuiram nas idéias e realização deste trabalho: Henry Vieira, Kaio Barbosa e Mikhail Ramalho; à coordenação e comissão do PPGCC-UFSCar, por me concederem a extensão ao prazo de defesa do Mestrado em função de problemas pessoais; aos amigos Erick Melo e Tiago Gaspar, que mesmo sem me conhecerem quando da minha chegada a São Carlos em 2009 acolheram-me com a gentileza que só as pessoas muito boas são capazes de demonstrar — sua ajuda é algo que nunca poderei agradecer o suficiente; à amiga Kamila Rios, pelas inúmeras vezes em que atuou como procuradora de meus interesses em todo o tempo que estive longe de São Carlos; e, por último, mas não menos importante, ao meu orientador: teria que escrever um capítulo deste trabalho só para agradecê-lo e dizer o quanto o admiro, tanto profissionalmente quanto pessoalmente; sou-lhe grato por tudo: pela oportunidade de ser seu aluno, pela orientação, pelos conselhos, pela preocupação e paciência com os meus problemas, pela amizade, estímulos e um longo etc. — devo a ele este trabalho e a ele dedico-o também.

♥

Necessito de um parágrafo à parte, especial, para homenagear aqui alguém mais do que especial na minha vida: meu muito obrigado à minha mulher, Paula Póvoas, amor da minha vida, por tudo. Quero lhe agradecer por toda dedicação e amor, pelo suporte, paciência, companheirismo, otimismo, sorrisos, alegria e pelo ânimo infundido quando as dificuldades e problemas pareciam encobrir tudo o mais. E, tão importante quanto tudo isso: por nunca ter deixado que eu esquecesse de que Deus olha por todos nós. Este trabalho é dedicado a você, meu amor.

*“Before software can be reusable it first has to be usable.”*

Ralph Johnson

# RESUMO

Middleware para TV Digital interativa tem a função de prover um ambiente padronizado para a execução de aplicações no qual dependências e/ou diferenças de hardware/software são abstraídas, permitindo o desenvolvimento de aplicativos portáveis. O desenvolvimento completo de um middleware é uma tarefa complexa e que consome muitos recursos. Entretanto, técnicas de reúso de software podem ser aplicadas de forma a tornar o processo de desenvolvimento mais eficiente através da reutilização de artefatos de software de outras implementações existentes do mesmo middleware. Dentro desse contexto, um processo de refatoração (*refactoring*) pode ser utilizado para melhorar características como modularidade ou reusabilidade, de modo a facilitar o reúso de partes dessas implementações. Esta dissertação de mestrado apresenta uma abordagem de refatoração voltada para portabilidade como estratégia de reúso em implementações de novas instâncias de middleware para TV Digital para plataformas de hardware diferentes daquela para o qual foi originalmente desenvolvido. A abordagem proposta apoia-se em conceitos e técnicas de portabilidade de software, bem como em características de arquitetura e aspectos de implementação de middleware para TV Digital. Como estudo de caso de aplicação dessa abordagem, resultados de experimentações realizadas com uma implementação do middleware do sistema brasileiro de TV Digital são apresentados e discutidos.

**Palavras-chave:** TV Digital, Middleware, Reúso, Refatoração, Portabilidade

# ABSTRACT

Middleware for interactive digital TV systems are designed to provide convenient abstractions to overcome differences of hardware/software in order to improve the portability of applications. The development of a middleware implementation from scratch is a complex and costly task. To minimize this effort, software assets of existing implementations of these middleware can be reused through software reuse techniques. A refactoring process can be applied to existing middleware implementations to improve some software requirements such as modularity and reusability, in order to facilitate reuse of the middleware. This work presents an approach of refactoring focused on portability as a reuse strategy for the development of new instances of a digital TV middleware for new hardware platforms. The proposed approach is based on software portability concepts and construction characteristics of digital TV middleware. To evaluate the work, we present a case study with our experience of refactoring an implementation of the Brazilian digital TV system's middleware.

**Keywords:** Digital TV, Middleware, Reuse, Refactoring, Portability

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

**ABNT** – Associação Brasileira de Normas Técnicas
**ACAP** – *Advanced Common Application Platform*
**API** – *Application Programming Interface*
**ARIB** – *Association of Radio Industries and Businesses*
**ATSC** – *Advanced Television System Committee*
**BML** – *Broadcast Markup Language*
**DVB** – *Digital Video Broadcasting*
**GEM** – *Globally Executable MHP*
**HTML** – *Hypertext Markup Language*
**IP** – *Internet Protocol*
**ISDB** – *Integrated Services Digital Broadcasting*
**ITU-T** – *Standardization sector of International Telecommunication Union*
**JPEG** – *Joint Photographic Experts Group*
**JVM** – *Java Virtual Machine*
**MHP** – *Multimedia Home Platform*
**MIME** – *Multipurpose Internet Mail Extension*
**MNG** – *Multiple-image Network Graphics*
**MPEG** – *Moving Picture Experts Group*
**NCL** – *Nexted Context Language*
**NCM** – *Nexted Context Model*
**PNG** – *Portable Network Graphics*
**POSIX** – *Portable Operating System Interface*
**SBTVD** – Sistema Brasileiro de TV Digital
**TVD** – TV Digital
**TVDi** – TV Digital interativa
**UML** – Unified Modeling Language
**URI** – Uniform Resource Identifier
**XHTML** – *Extensible Hypertext Markup Language*

# SUMÁRIO

# 1 INTRODUÇÃO

## 1.1 Contexto

Esta dissertação de mestrado apresenta uma abordagem para reutilização de middleware para TV Digital interativa – TVDi no processo de desenvolvimento de outras versões desses mesmos middleware, mas destinadas a plataformas de hardware diferentes daquelas para as quais foram originalmente desenvolvidas.

A abordagem proposta apoia-se em conceitos e técnicas de portabilidade de software, bem como em características de arquitetura e aspectos de implementação de middleware para TVDi.

## 1.2 Motivação

Middleware para TVDi tem a função de prover um ambiente padronizado para a execução de aplicações no qual dependências e diferenças de hardware/software são abstraídas, permitindo o desenvolvimento de aplicativos portáveis[1] para TV.

O desenvolvimento completo de um middleware é uma tarefa complexa e que consome muitos recursos, de modo que do ponto de vista prático e/ou comercial nem sempre constitui-se em uma solução desejável (MORRIS; CHAIGNEAU, 2005)(SCHMIDT; BUSCHMANN, 2003).

Uma alternativa ao desenvolvimento de um novo middleware é a reutilização de artefatos de software de implementações já existentes do mesmo middleware. Historicamente, o reúso de software tem sido utilizado com o propósito de aumentar a qualidade e a produtividade no processo de desenvolvimento, economizando esforços de codificação, testes, etc (FRAKES; KANG, 2005)(KRUEGER, 1992).

[1] O termo portável, aqui, diz respeito a software que pode ser executado sem nenhuma modificação em qualquer equipamento que possua esses middleware embarcados.

A reutilização de código fonte é uma das principais formas de reúso[2], entretanto, frequentemente algum tipo de *especialização* (modificação) do código pode ser necessária antes que esse possa ser efetivamente reutilizado.

O processo de modificação ou adaptação do código fonte de um software para possibilitar seu uso em um ambiente diferente daquele para o qual foi originalmente desenvolvido (outra plataforma de hardware, por exemplo) é conhecido como porte. Como atesta (MOONEY, 1995), portar claramente é uma forma de reúso de software.

Na prática, nem sempre um software é construído para ser facilmente portável ou reutilizável — muitas vezes isso implica custos adicionais em um projeto; daí, a necessidade de adaptação quando se deseja portar um software para um hardware e/ou sistema operacional diferentes.

O esforço de adaptação necessário durante um processo de porte depende sempre de um *contexto de porte* (MOONEY, 2004) específico, que define as diferenças entre o ambiente original do software e o novo ambiente para o qual deseja-se realizar o seu porte. Diferenças de hardware, sistema operacional e disponibilidade no novo ambiente de outros software utilizados na implementação são exemplos de fatores que afetam a portabilidade de um software.

Uma implementação de middleware para TVDi que pudesse ser portada com pouco esforço — com poucas modificações/adaptações — para outras plataformas de hardware poderia representar um ganho real no desenvolvimento de versões desse middleware para essas plataformas.

Assim, dadas a heterogeneidade de plataformas computacionais que podem operar como TV Digital (TVs, celulares, etc.) e a diversidade de fabricantes existentes, a possibilidade de porte/reúso de uma determinada implementação de middleware, ou de parte dela, no desenvolvimento de um novo produto pode constituir-se em uma alternativa interessante.

Uma implementação já existente de um middleware para TVDi poderia ser modificada de forma a se isolar partes mais genéricas ou facilmente portáveis do código (dentro de um determinado contexto) de outras, com dependências específicas de hardware/software; isso poderia ser feito, por exemplo, através de um processo de reorganização interna do código fonte conhecido como refatoração (*refactoring*).

Refatoração é o processo de modificar a estrutura interna de um software através da aplicação de uma série de tansformações (ou refatorações). O objetivo é redistribuir classes, variáveis, métodos, etc., de modo a facilitar futuras adaptações e extensões, melhorando a qualidade do software (modularidade, reusabilidade, manutenibilidade, etc.) e contribuindo para

[2] O termo reúso será utilizado no texto como sinônimo para reúso de software.

sua evolução (MENS; TOURWé, 2004)(OPDYKE, 1992).

Portanto, a aplicação de um processo de refatoração com enfoque em portabilidade em uma implementação de middleware para TVDi poderia ser utilizada para criar uma versão relativamente mais portável dessa implementação (ou de parte dela) incrementando sua capacidade de reutilização e contribuindo para o desenvolvimento de novas versões desse middleware para outras plataformas de hardware.

## 1.3 Objetivos

### Objetivo Geral

Explorar uma abordagem de refatoração com foco em portabilidade para ser aplicada em implementações de middleware para TVDi que não tenham sido desenvolvidas com enfoque no reúso, de modo a facilitar seu reúso/porte para plataformas de hardware diferentes daquelas para as quais foram originalmente desenvolvidas.

### Objetivos Específicos

1. Explorar que partes de uma implementação de middleware para TVDi são mais adequadas para porte para outras plataformas de hardware.
2. Propor estratégias de refatoração dessas partes mais portáveis de um middleware para TVDi, de forma a adaptá-las para utilização no desenvolvimento de novas versões desse middleware para outras plataformas de hardware.

## 1.4 Metodologia

- Estudo sobre portabilidade de software, fatores que a influenciam e como medi-la.
- Estudo sobre middleware para TV Digital interativa, seus padrões, especificações, características de arquitetura e implementação.
- Com base nos estudos prévios sobre portabilidade e middleware para TVDi, realizada análise para determinar que partes de um middleware para TVDi são mais adequadas para um processo de porte para outras plataformas de hardware.
- Desenvolvimento de estratégias de refatoração de implementações de middleware para TVDi com foco em suas partes mais portáveis. O objetivo da refatoração deve ser a

criação de componentes reutilizáveis/portáveis para serem utilizados no desenvolvimento de novas versões de middleware para outras plataformas de hardware, estabelecendo dessa forma um modelo de reúso.

- Aplicação da abordagem de reúso proposta em uma implementação de um middleware para TVDi e avaliação de sua eficácia no desenvolvimento de novas versões desse middleware para outras plataformas de hardware.

## 1.5 Organização

O Capítulo 2 discute os conceitos e técnicas de reúso, refatoração e portabilidade de software pertinentes a este trabalho.

O Capítulo 3 explora os middleware para TV Digitial interativa, com foco nos seus padrões e características de construção.

O Capítulo 4 apresenta a abordagem de reúso proposta neste trabalho, enquanto o Capítulo 5 descreve um estudo de caso de aplicação dessa abordagem em uma implementação do middleware Ginga, do sistema brasileiro de TV Digital.

As conclusões do trabalho são apresentadas no Capítulo 6.

# 2 REÚSO, REFATORAÇÃO E PORTABILIDADE

Este capítulo apresenta uma síntese sobre reúso, refatoração e portabilidade de software. A Seção 2.1 revisa os fundamentos de reúso. A Seção 2.2 introduz refatoração, descreve seu processo básico e as transformações utilizadas nesse. A Seção 2.3 apresenta um estudo sobre portabilidade, fatores que a influenciam, como medi-la e estratégias de porte.

## 2.1 Reúso de Software

### 2.1.1 Conceito

Reúso significa desenvolver software através da reutilização de artefatos de software já existentes. Artefatos reutilizáveis podem ser código fonte, programas executáveis, bibliotecas, arquiteturas, etc.

Historicamente, o reúso de software tem sido utilizado com o propósito de aumentar a qualidade e a produtividade no processo de desenvolvimento, economizando esforços de codificação, testes, etc (FRAKES; KANG, 2005)(KRUEGER, 1992).

### 2.1.2 Processo e Técnicas

Independentemente da abordagem ou técnica utilizada, o reúso de software envolve três importantes atividades que compõe a base de qualquer processo de reúso: seleção, especialização e integração de artefatos.

1. *Seleção:* localizar, comparar e selecionar artefatos reutilizáveis;
2. *Especialização:* customizar, transformar, parametrizar e refinar artefatos;

3. *Integração:* combinar uma coleção de artefatos selecionados e especializados para formar um software completo.

A Tabela 2.1 lista alguns exemplos de técnicas de reúso. A reutilização de código fonte é uma das principais formas de reúso, entretanto, frequentemente algum tipo de modificação do código pode ser necessário antes que esse possa ser efetivamente reutilizado.

Tabela 2.1: Técnicas de reúso

| **Técnica** | **Descrição** |
|---|---|
| Linguagens de alto nível | C, C++, Java, Lisp, etc. |
| Fragmentos de código | Explorar fragmentos de software existentes e utilizá-los como parte de um novo projeto. |
| Componentes | Subsistemas, módulos, pacotes, bibliotecas, classes. |
| Esquemas | Algoritmos, estruturas de dados, padrões de projeto. |
| Geradores de aplicação | Traduzem especificações de linguagens de alto nível em programas executáveis (compiladores, etc.). |
| Arquiteturas | Frameworks e subsistemas que capturam a estrutura global de um sistema. Oportunidade de reutilizar como um todo esforços de projeto e implementação. |

Reúso significa também construir software para ser reutilizado — um exemplo é o conceito de *engenharia de domínio* ou *linha de produtos de software* (CLEMENTS; NORTHROP, 2002), no qual software para um determinado domínio de aplicação é construído através da combinação de artefatos reutilizáveis desenvolvidos especificamente para esse domínio. É importante considerar, entretanto, que existem custos associados ao esforço de se desenvolver software para reúso.

Uma outra técnica de reúso bastante comum é a utilização de middleware (tema do Capítulo 3) e interpretadores de programas, como por exemplo, os navegadores de internet (que interpretam aplicações escritas em XHTML).

### 2.1.3 Métricas

Para analisar e melhorar a reusabilidade de um software, é necessário medi-la de alguma forma. A suíte de métricas para software orientado a objeto proposta por Chidamber e Kemerer

(CHIDAMBER; KEMERER, 1994) contém algumas métricas que podem ser úteis para esse tipo de análise:

**DIT (Depth of Inheritance Tree)** indica o comprimento do caminho em uma hierarquia de classes entre uma subclasse qualquer e a classe base. Quanto maior o DIT para uma classe, maior tende a ser a capacidade de reúso de métodos herdados.

**NOC (Number of Children)** número de subclasses diretamente derivadas de uma classe. Quanto maior o valor de NOC, maior o reúso, uma vez que herança é uma forma de reúso. Por outro lado, valores excessivos de NOC podem indicar abstrações inadequadas.

**CBO (Coupling Between Object)** indica o grau de acoplamento entre classes. Fornece uma medida da independência de uma classe com relação às outras. Quanto menor o valor do CBO, maior tende a ser a facilidade de reúso para uma classe e, de forma geral, maior a modularidade do software.

Bahtia e Mann (BHATIA; MANN, 2008) propuseram uma forma simples de combinar essas três métricas para calcular o valor para a reusabilidade (R) de uma classe:

$$R = DIT + NOC - 0,5 * CBO \tag{2.1}$$

Bahtia e Mann sugeriram também utilizar como valor para a reusabilidade de um diagrama de classes ou projeto o maior valor de R entre todas as classes que compõe esse diagrama/projeto.

## 2.2 Refatoração

### 2.2.1 Conceito

Refatoração é o processo de modificar a estrutura interna de um software através da aplicação de uma série de tansformações (ou refatorações). O objetivo é redistribuir classes, variáveis, métodos, etc., de modo a facilitar futuras adaptações e extensões, melhorando a qualidade do software (modularidade, reusabilidade, manutenibilidade, etc.) e contribuindo para sua evolução (MENS; TOURWé, 2004)(OPDYKE, 1992).

Uma aplicação natural para refatoração é no contexto de reúso de software. Transformações podem ser aplicadas em um software para, por exemplo, melhorar sua modularidade e criar componentes reutilizáveis.

### 2.2.2 Processo

Apesar de haver algumas variações na sua descrição na literatura, um processo de refatoração pode ser visto como composto das seguintes etapas (KATAOKA, 2006)(MENS; TOURWé, 2004):

1. Identificar onde o software deve ser refatorado.
2. Escolher quais transformações devem ser aplicadas.
3. Aplicar as transformações.
4. Avaliar o efeito da refatoração no software (reusabilidade, portabilidade, etc.).

#### Identificação de *bad smells*

Refatorações devem ser aplicadas nas partes consideradas deficitárias de um sofware. Essas deficiências na implementação são conhecidas como *bad smells* (FOWLER; BECK, 1999). Exemplos de *bad smells* são duplicação de código, métodos muito longos, mistura de lógica de aplicação com lógica de apresentação, etc.

#### Transformações

Uma vez identificados os *bad smells*, a próxima etapa é a escolha de quais refatorações devem ser aplicadas. O trabalho de Opdyke (OPDYKE, 1992) é considerado o primeiro sobre refatoração e introduziu uma série de transformações primitivas que podem ser utilizadas individualmente ou em conjunto para compor transformações mais sofisticadas.

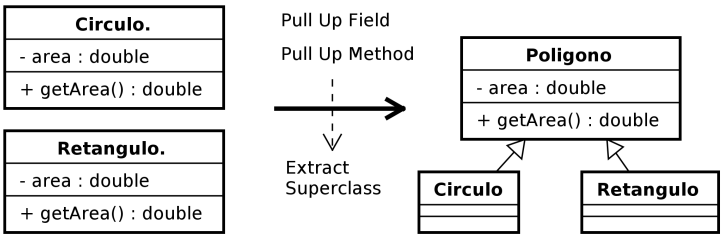


Figura 2.1: Exemplo de refatoração

A Figura 2.1 ilustra um exemplo de aplicação da transformação Extract Superclass (FOWLER; BECK, 1999), para criar uma nova hierarquia de classes do tipo `Poligono` a partir de duas classes existentes (`Circulo` e `Retangulo`).

A aplicação da refatoração permitiu unificar atributos e métodos de duas classes diferentes, eliminando a duplicação de código e criando uma nova classe reutilizável, que pode ser estendida para criar novas classes de polígono. É importante observar que a transformação foi composta a partir das transformações primitivas Pull Up Field e Pull Up Method (OPDYKE, 1992), que moveram, respectivamente, os atributos `area` e os métodos `getArea()` das classes originais para a nova classe `Poligono`.

Um exemplo de transformação inversa a Extract Superclass é a transformação Collapse Hierarchy (FOWLER; BECK, 1999), que reduz uma hierarquia de classes quando se julga que algumas subclasses são dispensáveis e podem ser descartadas, simplificando o código.

#### Aplicação das transformações

Refatorações podem ser aplicadas de forma manual, semiautomática ou automática. Um dos objetivos do trabalho original de Opdyke foi descrever refatorações como um conjunto de regras e condições para sua aplicação, de forma que pudessem ser automatizadas. Editores de texto, *scripts* e ambientes gráficos integrados podem ser utilizados em abordagens semiautomáticas e/ou manuais.

#### Avaliação da refatoração

Um processo de refatoração deve ter um objetivo definido, como por exemplo, melhorar a modularidade ou a portabilidade de um software. Portanto, é fundamental o uso de métricas para se avaliar o efeito do processo de transformação. Um exemplo pode ser visto no trabalho de Kataoka (KATAOKA, 2006)(KATAOKA et al., 2002), que desenvolveu métodos para avaliar quantitativamente os efeitos de refatorações na manutenibilidade de um software. Outro exemplo são as métricas relacionadas a reusabilidade, que foram mostradas na Seção 2.1.

### 2.2.3 Padrões de Projeto

Padrões de projeto (*design patterns*) são soluções genéricas (e reutilizáveis) para problemas comuns que ocorrem no desenvolvimento de software. Padrões de projeto são largamente utilizados no projeto e evolução de software orientado a objeto, constituindo uma ferramenta fundamental para documentar e implementar soluções recorrentes e já testadas (GAMMA et al., 1995).

A Figura 2.2 ilustra um exemplo do padrão de projeto Adapter. O objetivo desse padrão é converter a interface (métodos) de uma classe (`Component`) em outra interface que uma

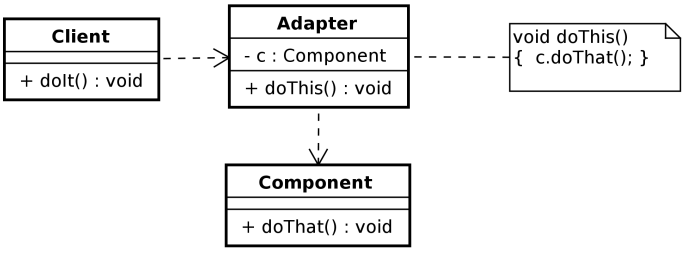


Figura 2.2: Padrão de projeto Adapter

classe cliente da primeira (`Client`) necessita. Dessa forma, a classe `Adapter` permite que duas classes com interfaces incompatíveis trabalhem juntas.

Um outro exemplo de padrão de projeto conhecido é mostrado na Figura 2.3. O padrão Proxy (também conhecido como Wrapper ou Surrogate) descreve uma classe/objeto (`Proxy`) que encapsula outra classe/objeto (`RealResource`), de forma a controlar o acesso a essa (GAMMA et al., 1995).

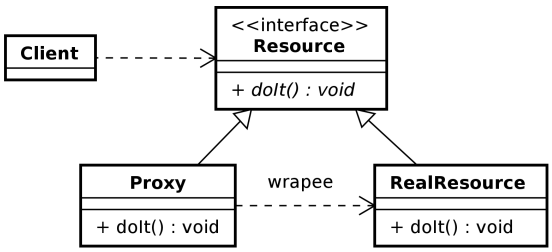


Figura 2.3: Padrão de projeto Proxy

O padrão Proxy tem diversas aplicações, entre as quais, abstrair objetos residentes em outro espaço de endereçamento, criar objetos por demanda (por exemplo, objetos que exigem bastante processamento, como gráficos), controle de acesso, etc.

Na prática, muitos processos de refatoração são realizados somente com a introdução de padrões de projeto, como pode ser visto em (KERIEVSKY, 2005)(TOKUDA; BATORY, 2001). Esses trabalhos mostram também como alguns padrões de projeto podem ser obtidos a partir da aplicação conjunta de transformações mais simples, como as introduzidas nos trabalhos de Opdyke (OPDYKE, 1992) e Fowler (FOWLER; BECK, 1999).

### 2.2.4 Catálogos de Refatorações

Transformações e padrões de projeto utilizados para refatoração compõe catálogos que encontram-se disponíveis na literatura e que podem ser utilizados como referência na implementação de soluções úteis e já testadas.

As transformações introduzidas por Opdyke (OPDYKE, 1992) compõe um catálogo básico de refatorações primitivas que são utilizadas por outros autores, como Fowler (FOWLER; BECK, 1999), que estende essa lista de refatorações e as utiliza para compor refatorações de mais alto nível, como mostrado na Seção 2.2.2. O catálogo de Fowler é constantemente atualizado e pode ser consultado em `http://refactoring.com/catalog/index.html`.

Um outro catálogo útil é o de Kerievsky (KERIEVSKY, 2005), que combina os catálogos de padrões de projeto (GAMMA et al., 1995) e de Fowler para construir refatorações de nível ainda mais alto, que são motivadas e dirigidas pela introdução de padrões de projeto. A versão mais recente desse catálogo pode ser consultada em `http://industriallogic.com/xp/refactoring/catalog.html`.

É importante observar que os catálogos de refatoração não têm a pretensão de esgotar todas as possibilidades de transformações. A literatura mostra que constantemente surgem novas transformações, como pode ser visto em (TAHVILDARI; KONTOGIANNIS, 2002)(TOKUDA; BATORY, 2001).

## 2.3 Portabilidade

### 2.3.1 Conceito

Portabilidade refere-se à capacidade de uma unidade de software poder ser adaptada para uso em um novo ambiente a um custo inferior ao de desenvolvê-la completamente. Como atesta Mooney (MOONEY, 1995), portar claramente é uma forma de reúso de software.

Unidades de software podem ser aplicativos, componentes ou até mesmo sistemas inteiros, tanto em nível de código executável (ou binário) quanto de código fonte. Por ambiente entende-se a coleção de elementos externos com os quais o software interage: plataforma de hardware, sistema operacional, outros software, etc.

Adaptação refere-se às (possíveis) modificações em uma unidade de software de forma que ela possa ser transportada fisicamente e executada em um ambiente diferente do seu de origem. Uma unidade de software é perfeitamente portável se o custo do processo de porte é zero (isto é, nenhuma adaptação é necessária).

### 2.3.2 Portabilidade Binária e de Código Fonte

Existem duas formas de portabilidade: a primeira é a utilização de um software existente sem nenhuma modificação no novo ambiente — conhecida como *portabilidade binária*; a segunda (que está mais próxima do conceito tradicional de reúso) é a *portabilidade de código fonte*, na qual modificações em um software são necessárias, de modo a adaptá-lo para uso no novo ambiente.

Na portabilidade de código fonte, técnicas de reúso e refatoração podem ser aplicadas no processo de adaptação do software para o ambiente-alvo; contudo, o custo de adaptação deve ser avaliado, de forma a se decidir se o melhor é portar o software ou desenvolvê-lo novamente.

### 2.3.3 Contexto de Porte e Grau de Portabilidade

O esforço de adaptação necessário durante um processo de porte depende sempre de um *contexto de porte* (MOONEY, 2004), que define as diferenças entre o ambiente original do software e o novo ambiente para o qual deseja-se realizar o seu porte.

Um conceito fundamental é o que Mooney (MOONEY, 1995) define como *grau de portabilidade*, que é uma função dos custos dos processos de porte e desenvolvimento de um software para um determinado ambiente. Assim, portabilidade não é uma característica absoluta para um software e depende sempre de um contexto de porte específico.

A Equação 2.2 mostra uma maneira simples e intuitiva para avaliar o grau de portabilidade (DP) de um software em um determinado contexto:

$$DP = 1 - (C_{port}/C_{rdev}) \tag{2.2}$$

*Cport* significa o custo do processo de porte e *Crdev* o custo de desenvolver o software para o novo ambiente. Se $DP \geq 0$, portar o software deve ser mais vantajoso do que redesenvolvê-lo.

### 2.3.4 Dependências Externas e Controle de Interfaces

Diversos fatores afetam o grau de portabilidade de um software. Embora a identificação precisa desses fatores (e o peso de cada um) dependa de cada contexto de porte, os principais são (HAKUTA; OHMINAMI, 1997)(MOONEY, 2004):

a) Diferenças de hardware (processador, periféricos);

b) Diferenças de sistema operacional (serviços e funções de sistema);

c) Linguagem de programação utilizada;

d) Dependências externas (bibliotecas, outros software, etc.);

e) Fatores humanos (conhecimento, experiência com porte);

f) Fatores ambientais (ambientes de desenvolvimento e testes);

g) O próprio software (extensão, o quanto foi projetado para ser portável, etc).

Um dos fatores que podem limitar bastante a portabilidade é a *dependência externa* de outros software utilizados no desenvolvimento de um programa. Bibliotecas, pacotes (conjunto de bibliotecas) e outros software devem ser mapeados e analisados quando se considera um processo de porte, uma vez que o uso desses no novo ambiente talvez não seja adequado ou possível (inexistência de código fonte ou binários, dificuldade de porte, etc.).

De forma geral, quanto menor o número de dependências externas, maior tende a ser o grau de portabilidade para um software (HAKUTA; OHMINAMI, 1997)(MOONEY, 2004)(MOONEY, 1995).

Um princípio-chave para portabilidade é a identificação e *controle das interfaces* externas de um software (Figura 2.4). Essas interfaces incluem as APIs (*Application Programming Interfaces*) do sistema operacional (SO) e de todas as dependências externas.

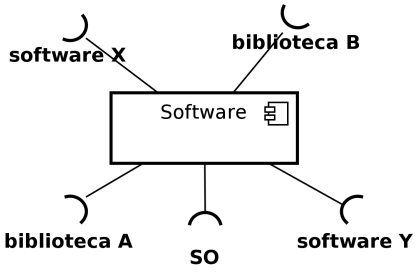


Figura 2.4: Interfaces externas de um software

Isolar/restringir o uso dessas interfaces a partes específicas e facilmente identificáveis em um software contribui para facilitar futuros processos de porte, uma vez que muitos problemas de portabilidade são causados por incompatibilidades entre o software e o ambiente externo.

Mooney (MOONEY, 2004) cita três estratégias de controle de interfaces que podem ser utilizadas tanto no desenvolvimento quanto no porte de um software:

- *Padronização:* identificar padrões e domínios de aplicação existentes e largamente utilizados. Por exemplo, compiladores de linguagens altamente difundidas como C/C++ estão disponíveis em vários ambientes; isso contribui para simplificar futuros processos de porte de software desenvolvido nessas linguagens.

- *Porte de dependências:* se os componentes do outro lado das interfaces podem ser reimplementados ou modificados em vários ambientes, então essa pode ser uma estratégia válida para portabilidade, uma vez que mantém a consistência das interfaces sem a necessidade de modificação do software a ser portado.

- *Tradução:* se as interfaces utilizadas por um software são diferentes daquelas presentes no novo ambiente, adaptações serão necessárias. Essas adaptações incluem modificações em um dos lados (estratégia anterior), intermediações feitas por outros software, etc.

# 3 MIDDLEWARE PARA TV DIGITAL INTERATIVA

Este capítulo apresenta um breve estudo sobre middleware para TVDi. A Seção 3.1 apresenta a visão geral de um sistema de TV Digital e os principais sistemas existentes. A Seção 3.2 descreve conceitos, padrões, normas e características de construção utilizados no desenvolvimento de middleware para TVDi.

## 3.1 TV Digital

### 3.1.1 Infraestrutura de Sistema

Dentre os vários padrões de TV Digital existentes, destacam-se os trabalhos realizados nas últimas duas décadas nos Estados Unidos, Europa e Japão que deram origem, respectivamente, aos sistemas ATSC, DVB e ISDB (WU et al., 2006)(MORRIS; CHAIGNEAU, 2005).

Mais recentemente, o Brasil também desenvolveu seu próprio padrão de TV Digital terrestre conhecido como SBTVD ou ISDB-TB, cujas especificações foram publicadas pela ABNT na série de normas NBR 15601 a 15608.

A arquitetura de um sistema de TV Digital é composta basicamente por três subsistemas: (i) Codificação e Compressão, (ii) Empacotamento/Multiplexação de dados e (iii) Transmissão – que varia de acordo com o meio utilizado para transporte das informações (cabo, satélite, terrestre ou redes IP).

A Figura 3.1 mostra a interligação dos subsistemas de um sistema de TV Digital típico. O subsistema de Codificação e Compressão utiliza codificações baseadas nos padrões MPEG para áudio e vídeo. O subsistema de Empacotamento utiliza o padrão MPEG-2 *System* (ISO/IEC, 2000) e o subsistema de Transmissão utiliza algum esquema de modulação digital adequado ao meio utilizado.

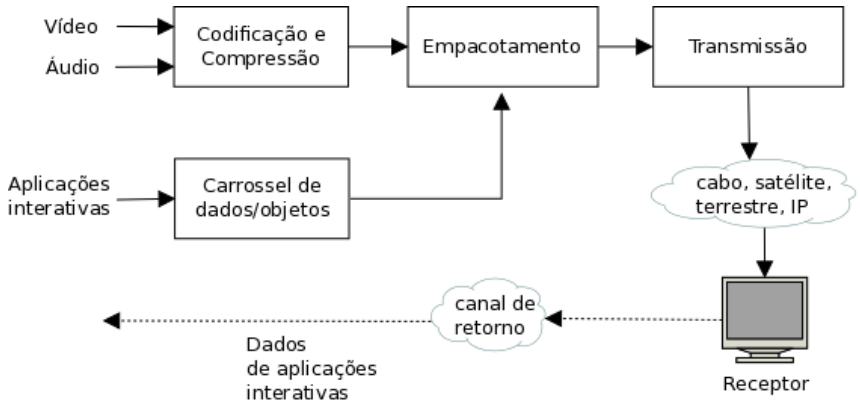


Figura 3.1: Arquitetura de um sistema de TV Digital

### 3.1.2 TV Digital Interativa

Além da qualidade superior de vídeo e áudio, a tecnologia de TV Digital proporciona outros atrativos. Um desses, é a possibilidade do telespectador interagir com aplicações enviadas pelas emissoras de TV, caracterizando o que se conhece por TV Digital interativa.

As aplicações interativas (jogos, comércio, serviços governamentais, etc.) são enviadas pelas emissoras pelo mesmo canal utilizado para a transmissão dos programas de TV (Figura 3.1) e podem ou não ter alguma relação com esses.

Como pode ser visto em (CRINON et al., 2006)(MORRIS; CHAIGNEAU, 2005), um mecanismo conhecido como carrossel de dados/objetos é responsável por fragmentar os arquivos que compõem as aplicações de forma que essas possam ser adequadamente empacotadas e transmitidas.

Uma aplicação interativa eventualmente pode fazer uso de um canal de comunicação, denominado canal de retorno (Figura 3.1), implementado através de internet, rede de telefonia com ou sem fio, WiMAX, etc., para trocar dados com uma aplicação ou serviço remoto. A disponibilidade do canal de retorno depende da configuração do equipamento receptor.

Para que um receptor possa executar uma aplicação interativa, é necessário que esse possua embarcado o middleware padrão para o sistema de TV Digital. Como será visto na próxima seção, sem um middleware ficaria muito mais difícil a execução de aplicações interativas em um receptor.

## 3.2 Middleware para TV Digital

### 3.2.1 Conceito

As aplicações para TVDi são enviadas pelas emissoras juntamente com os programas transmitidos. Depois de recebê-las, o equipamento receptor de TV pode então executá-las, de acordo com a conveniência do usuário.

Devido à grande variedade de dispositivos receptores, com suas respectivas particularidades de hardware e software, foi necessário que os sistemas de TVDi adotassem o uso de middleware como plataforma de execução para essas aplicações.

Um middleware (Figura 3.2) é uma camada de software intermediária entre aplicações e sua plataforma de execução (hardware, sistema operacional, etc.) que abstrai características específicas de hardware e software dessa plataforma através da disponibilização de serviços e de um conjunto de APIs para acessá-los.

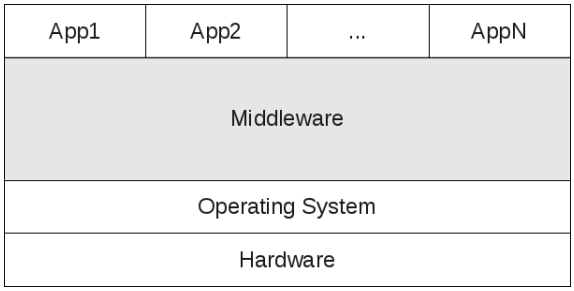


Figura 3.2: Middleware

O principal propósito de um middleware é prover um ambiente que possibilite a execução de aplicações de forma transparente em múltiplas plataformas de hardware/software. Um exemplo de middleware bem conhecido é a JVM, utilizada para executar programas escritos em Java.

Sem um middleware, seria muito mais difícil a execução de aplicações interativas em um receptor de TVDi, uma vez que as emissoras de TV não podem prever o tipo e as características dos receptores (hardware, sistema operacional, etc.). Mesmo que isso fosse possível/viável, ainda assim seria necessário o envio de diversas versões de uma mesma aplicação para atender cada tipo de ambiente nativo encontrado no universo dos fabricantes de receptores de TVDi.

### 3.2.2 Padrões

Os principais padrões de middleware para TVDi existentes (Tabela 3.1) — europeu/MHP, americano/ACAP, japonês/ARIB, brasileiro/Ginga, ITU-T/J.200 — (PIESING, 2006)(MOR-

RIS; CHAIGNEAU, 2005)(ITU-T, 2001)(SOARES; RODRIGUES; MORENO, 2007)(FILHO; LEITE; BATISTA, 2007) especificam suporte para o desenvolvimento de aplicações utilizando dois paradigmas de programação: imperativo e declarativo. Java é a linguagem imperativa mais utilizada, enquanto que as aplicações declarativas são desenvolvidas utilizando linguagens de marcação baseadas em XML.

Tabela 3.1: Padrões de middleware para TV Digital

| **Sistema** | **Origem** | **Middleware** |
|---|---|---|
| ATSC | Estados Unidos | ACAP |
| DVB | Europa | MHP |
| ISDB | Japão | ARIB |
| ISDB-TB | Brasil | Ginga |

### MHP e GEM

O sistema europeu DVB utiliza como middleware o MHP (PIESING, 2006)(MORRIS; CHAIGNEAU, 2005), que permite a execução de aplicações imperativas escritas em Java. As aplicações declarativas são escritas em DVB-HTML, que é baseada em XHTML e possui suporte a ECMAScript.

Para permitir a execução de aplicações MHP em outros sistemas de TV Digital, foi proposta uma harmonização de várias especificações já existentes, chamada GEM, que corresponde a um subconjunto do MHP.

O sistema americano ATSC e o japonês ISDB adotaram o GEM como base para seus respectivos middleware, ACAP e ARIB. Para as aplicações declarativas, o ACAP utiliza a linguagem ACAP-X e o ARIB a linguagem BML, ambas baseadas em XHTML.

### ITU-T J.200

De forma geral, os middleware para TVDi são compatíveis com a recomendação ITU-T J.200 (ITU-T, 2001), que foi desenvolvida com o intuito de colaborar na harmonização de todos os padrões existentes e na especificação de padrões globais para middleware para TVDi.

Além da J.200, o ITU-T possui duas outras recomendações que são utilizadas em conjunto com essa: as ITU-T J.201 (ITU-T, 2008) e J.202 (ITU-T, 2003), que especificam, respectivamente, os padrões utilizados para suporte à aplicações declarativas e imperativas.

### 3.2.3 Arquitetura e Implementação

A recomendação ITU-T J.200 descreve a arquitetura geral de um middleware para TVDi (Figura 3.3), dividindo-a em três partes principais: subsistema declarativo (*presentation engine*), subsistema imperativo (*execution engine*) e um subsistema comum (que serve aos primeiros), através do qual é disponibilizado o acesso a recursos e funções específicas de um receptor de TVDi (funções gráficas, decodificação e exibição de vídeo, áudio e imagens, interação com usuário via controle remoto, informações sobre programas de TV, etc.).

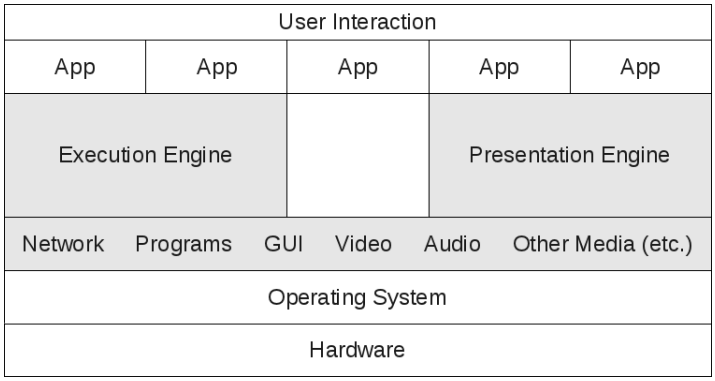


Figura 3.3: Middleware para TVDi (ITU-T J.200)

A arquitetura em camadas dos middleware para TVDi é uma característica própria dos middleware em geral. Como é sabido, esse tipo de arquitetura e a organização em subsistemas são fatores que contribuem positivamente para características como reusabilidade e portabilidade (SCHMIDT; BUSCHMANN, 2003)(BASS; CLEMENTS; KAZMAN, 2003)(GARLAN; SHAW, 1993).

É importante observar também alguns aspectos de implementação relevantes para middleware para TVDi:

- As plataformas embarcadas comumente utilizadas em produtos de TV Digital possuem recursos limitados. A execução de aplicações interativas sem nenhum controle pode esgotar os recursos disponíveis na plataforma, tais como memória e processamento. Portanto, *controlar e gerenciar o acesso a recursos* como os decodificadores e exibidores de mídias (áudio, vídeo, imagens, etc.) é fundamental em uma implementação de middleware.

- Um middleware é um software complexo, cujo desenvolvimento tende a ser custoso e demorado. Portanto, é desejável que eles sejam desenvolvidos pensando-se em alguma forma de reúso para ao menos parte de sua implementação. O uso da arquitetura apresentada, de frameworks, componentes e de linguagens como C, C++ e Java — bastante

comuns no desenvolvimento desse tipo de software — contribui positivamente para isso (MORRIS; CHAIGNEAU, 2005)(SCHMIDT; BUSCHMANN, 2003).

### 3.2.4 Ginga

Ginga é o middleware do sistema brasileiro de TV Digital (ISDB-TB). A arquitetura do Ginga (ABNT, 2011)(SOARES; RODRIGUES; MORENO, 2007)(FILHO; LEITE; BATISTA, 2007) é compatível com a recomendação ITU-T J.200 (Seção 3.2.3), conforme mostrado na Figura 3.4.

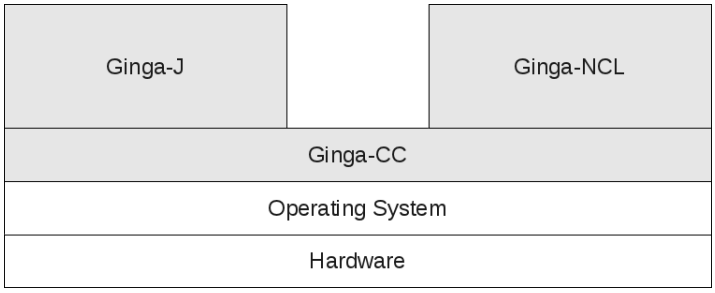


Figura 3.4: Middleware Ginga

O Ginga é composto basicamente de três subsistemas: Ginga-J, Ginga-NCL e Ginga-CC. Ginga-J é o subsistema imperativo, responsável pela execução de aplicações escritas em Java. Ginga-NCL é o subsistema declarativo, responsável pela execução de aplicações escritas na linguagem NCL. O subsistema comum, denominado Ginga-CC ou *Common Core* (CC), provê o acesso aos recursos do receptor de TVDi.

Além de aplicações escritas puramente em NCL, o Ginga fornece suporte também à execução de *scripts* escritos em Lua, que são referenciados/utilizados por documentos NCL. A integração NCL/Lua no Ginga pode ser vista em (SOARES; BARBOSA, 2009)(ABNT, 2011).

A Listagem 3.1 mostra um exemplo de aplicação interativa para o Ginga escrita na linguagem NCL — o funcionamento da aplicação é simples: assim que ela é iniciada (elemento `<port>`, linha 32), uma imagem representando o ícone da aplicação (elemento `<media>`, linha 34) é colocada no canto superior direito da tela (as regiões de tela utilizadas na aplicação são determinadas pelos elementos `<regionBase>` e `<region>`, linhas 4 a 10).

Quando a tecla ENTER do controle remoto é pressionada, a aplicação aguarda 2 segundos e em seguida executa as seguintes ações: retira da tela o ícone da aplicação, faz o redimensionamento da região de vídeo do programa de TV (elemento `<media>` do tipo `sbtvd-ts://video`, linhas 35 a 37) e coloca na tela o conteúdo de um arquivo texto (elemento `<media>`, linhas 38 a 40).

Listing 3.1: Aplicação NCL para o Ginga

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ncl id="selection" xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
   <head>
      <regionBase>
         <region id="rbTV" width="100%" height="100%" device="systemScreen(0)">
            <region id="rTela" left="0%" top="0%" width="100%" height="100%"/>
            <region id="rIcon" right="5%" top="5%" width="10%" height="10%"/>
            <region id="rText" left="10%" top="50%" width="80%" height="40%"/>
         </region>
      </regionBase>

      <descriptorBase>
         <descriptor id="dTela" region="rTela"/>
         <descriptor id="dIcon" region="rIcon"/>
         <descriptor id="dText" region="rText"/>
      </descriptorBase>

      <connectorBase>
         <causalConnector id="onKeySelectionStopSetStart">
            <connectorParam name="keyCode"/>
            <simpleCondition role="onSelection" key="$keyCode" delay="2s"/>
            <compoundAction operator="par">
               <simpleAction role="stop" max="unbounded"/>
               <simpleAction role="set" value="$newValue" max="unbounded"/>
               <simpleAction role="start" max="unbounded"/>
            </compoundAction>
         </causalConnector>
      </connectorBase>
   </head>

   <body>
      <port id="entrada" component="icon"/>

      <media descriptor="dIcon" id="icon" src="icon.jpg"/>
      <media descriptor="dTela" id="prog" src="sbtvd-ts://video" >
         <property name="bounds"/>
      </media>
      <media descriptor="dText" id="news" src="news.txt" >
         <property name ="fontColor" value="white"/>
      </media>

      <link xconnector="onKeySelectionStopSetStart">
         <bind role="onSelection" component="icon">
            <bindParam name="keyCode" value="ENTER"/>
         </bind>
         <bind role="stop" component="icon"/>
         <bind role="set" component="prog" interface="bounds">
            <bindParam name="newValue" value="10%,5%,40%,40%"/>
         </bind>
         <bind role="start" component="news"/>
      </link>
   </body>
</ncl>
```

Os eventos descritos são um exemplo do tipo de interação com o usuário e sincronização de mídias possível em uma aplicação NCL (esse comportamento é definido pelos elementos `<connectorBase>` e `<causalConnector>`, linhas 18 a 28, e implementado no elemento `<link>`, linhas 42 a 51).

Mais detalhes sobre a linguagem NCL e seu uso em aplicações para o Ginga podem ser vistos em (ABNT, 2011) e (SOARES; BARBOSA, 2009).

# 4 REFATORAÇÃO DE MIDDLEWARE PARA PORTE

## 4.1 Visão Geral

Nos capítulos anteriores foram apresentados os conceitos de middleware para TV Digital interativa, reúso, refatoração e portabilidade de software pertinentes a este trabalho.

A partir desses estudos, este capítulo propõe uma abordagem de refatoração com enfoque em portabilidade para ser aplicada em implementações de middleware para TVDi. O objetivo é possibilitar o reúso de parte dessas implementações no desenvolvimento de outras versões desse mesmo middleware, mas destinadas a outras plataformas de hardware.

O foco da proposta são implementações de middleware para TVDi que não tenham sido desenvolvidas com enfoque no reúso ou cuja reutilização seja difícil no contexto de mudança de plataforma de hardware.

A abordagem proposta divide-se em duas partes: (i) Análise de que partes de um middleware para TVDi são mais adequadas para reúso/porte em outras plataformas de hardware (Seção 4.2); e (ii) Desenvolvimento de um modelo de reúso baseado na adaptação/refatoração dessas partes para facilitar seu porte para outras plataformas (Seção 4.3).

## 4.2 Reutilizando Middleware

### 4.2.1 Motivação para o Reúso

Conforme sugerido na Seção 3.2.3, o desenvolvimento de um middleware para TVDi é uma tarefa complexa que pode demandar muitos recursos. Tal empreendimento sem considerar projetos semelhantes pode não ser uma decisão economicamente viável.

Uma opção para o desenvolvimento de um novo middleware é a reutilização do código

fonte de implementações já existentes do mesmo middleware, conforme visto no Capítulo 2. Entretanto, nem sempre um software é construído para ser facilmente reutilizado — muitas vezes o desenvolvimento para reúso implica custos adicionais em um projeto. Nesses casos, um esforço pode ser necessário quando se deseja, por exemplo, portar um software para um ambiente diferente.

### 4.2.2 Porte como Estratégia de Reúso

Mesmo que uma implementação de middleware para TVDi não tenha sido desenvolvida com foco em reúso, é possível tirar proveito de certas características de construção desse tipo de software que podem favorecer o porte dessa implementação para outras plataformas de hardware.

Conforme visto na Seção 2.3, portar é uma forma de reúso de software. A Figura 4.1 mostra um gráfico de *dependências* contendo alguns dos fatores que afetam a portabilidade.

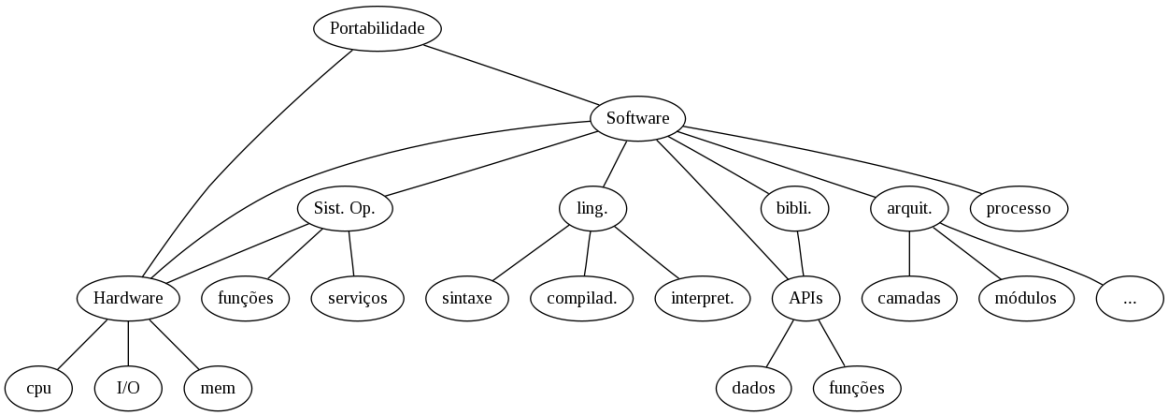


Figura 4.1: Gráfico de dependências para portabilidade

O gráfico mostra também as interdependências existentes entre alguns fatores. Por exemplo, o fator linguagem de programação (ling.) é diretamente afetado pela disponibilidade ou não de compiladores, interpretadores, etc, para essa linguagem em determinado ambiente. Outro exemplo ilustrado é a dependência de um software com relação ao sistema operacional (Sist. Op.) para o qual foi desenvolvido, devido ao uso de funções e serviços de sistema.

Software embarcado, como um middleware para TVDi, é comumente desenvolvido em linguagens de programação como C, C++, Java, etc., conforme visto no Capítulo 3. Essas linguagens são escolhidas em função de questões como desempenho, uso eficiente de recursos (que tendem a ser limitados em plataformas de TVDi) e por possuirem ampla cobertura de compiladores (C/C++) e interpretadores (JVM) para os mais diversos ambientes.

Assim, software escrito nessas linguagens tende naturalmente a possuir certa portabilidade dentro do contexto de mudança de hardware (uma vez que estejam disponíveis os compiladores e interpretadores necessários).

No caso de C/C++, outro fator importante é o sistema operacional alvo (caso exista) da implementação. Sistemas operacionais encapsulam e abstraem em boa medida discrepâncias de hardware, disponibilizando serviços e APIs para acesso a esses (MOONEY, 2004)(MOONEY, 1995).

Portanto, mesmo num contexto de mudança de hardware, a manutenção do SO e o uso das linguagens citadas, contribuem para facilitar o porte de um software. Contudo, existem fatores que afetam a portabilidade e que independem do SO e linguagem utilizados. Como visto na Seção 2.3.4, as dependências externas de outros software é um fator importante a ser considerado em qualquer processo de porte.

Dependências específicas de plataforma podem dificultar ou mesmo inviabilizar o porte de um software. A próxima seção trata desse aspecto no contexto de middleware para TVDi.

### 4.2.3 Avaliando Possibilidades de Porte

Um middleware para TVDi faz uso de recursos de hardware muito específicos de uma plataforma, como por exemplo decodificação de áudio e vídeo. O acesso a essas funcionalidades é disponibilizado normalmente através de APIs proprietárias cujo código fonte muitas vezes não está acessível ou é de difícil adaptação para uma outra plataforma, visto que cada fabricante de dispositivos possui sua própria padronização.

Assim, devido a dependências específicas, portar uma implementação de middleware para TVDi na sua totalidade para uma plataforma de hardware diferente da original pode ser uma tarefa muito difícil ou mesmo impossível.

Uma maneira de isolar dependências específicas de uma plataforma é criar abstrações através da utilização de camadas de software, tal como na arquitetura de middleware para TVDi (Figura 3.3). Esse tipo de arquitetura, em conjunto com os fatores vistos na seção anterior, favorece a criação de software reutilizável/portável, conforme visto na Seção 3.2.1.

A abordagem de construção em camadas é bastante utilizada na prática, fato que pode ser comprovado através de diversos trabalhos sobre desenvolvimento de middleware para TVDi, tais como (GAO; WANG; NI, 2009), (CRUZ; MORENO; SOARES, 2008), (FERREIRA et al., 2010) e (ELIAS et al., 2004).

Dessa forma, é razoável supor que seja possível portar ao menos parte de um middleware para TVDi para outra plataforma de hardware. Quais partes são mais adequadas para essa tarefa depende da implementação e do contexto de porte, mas é possível tirar algumas conclusões a respeito considerando-se as características de construção desse tipo de middleware:

- Supondo um mínimo de organização na definição da arquitetura e na implementação, é provável que o middleware possua, ao menos, uma camada de software intermediária entre os subsistemas declarativo/imperativo e os recursos da plataforma/sistema operacional — o subsistema comum (que disponibiliza o acesso aos recursos do equipamento de TVDi), tal como visto na Seção 3.2.3.

- Independentemente do número de camadas utilizadas na organização da arquitetura do middleware, é mais provável que o uso direto dos recursos específicos de uma plataforma estejam concentrados nas camadas mais internas do middleware, isto é, aquelas mais próximas do sistema operacional/hardware, como, por exemplo, o subsistema comum (tal como pode ser visto em (GAO; WANG; NI, 2009)(CRUZ; MORENO; SOARES, 2008)(FERREIRA et al., 2010)(ELIAS et al., 2004)).

- É possível que o subsistema comum seja composto de muitas camadas de abstração, de modo que se tenha uma implementação altamente portável. Entretanto, deve haver um limite para isso, uma vez que excesso de abstrações pode levar a problemas de desempenho e, no caso de middleware para TVDi, a replicar em software funções que são melhores desempenhadas pelo hardware específico (MORRIS; CHAIGNEAU, 2005)(BASS; CLEMENTS; KAZMAN, 2003).

Portanto, parece provável que as partes de um middleware para TVDi mais adequadas para um possível reúso/porte encontram-se nas camadas mais externas da arquitetura — isto é, aquelas mais distantes do hardware, como os subsistemas declarativo e imperativo.

## 4.3 Refatorando Middleware

### 4.3.1 Modelo de Reúso

Uma vez que, em tese, os subsistemas declarativo e imperativo de um middleware para TVDi são candidatos melhores a reúso do que o subsistema comum, a abordagem de reúso proposta será baseada na reutilização dessas partes. O quanto de uma implementação de um

desses subsistemas pode ser reutilizado, depende da implementação e também do contexto de reúso.

É possível garimpar uma implementação de forma a se identificar componentes (ou criá-los, através de um processo de adaptação) com o objetivo de utilizá-los em novas implementações.

Uma outra possibilidade é tentar portar um dos subsistemas (imperativo ou declarativo) para um novo ambiente. Nesse caso, o esforço empregado nesse processo dependerá do contexto de porte, conforme visto na Seção 2.3.

Este trabalho propõe o desenvolvimento de novas instâncias de um middleware para TVDi através da criação e porte de componentes reutilizáveis desenvolvidos a partir da adaptação dos subsistemas declarativo e imperativo originais para outras plataformas de hardware. Esse processo é composto de três atividades principais (Figura 4.2):

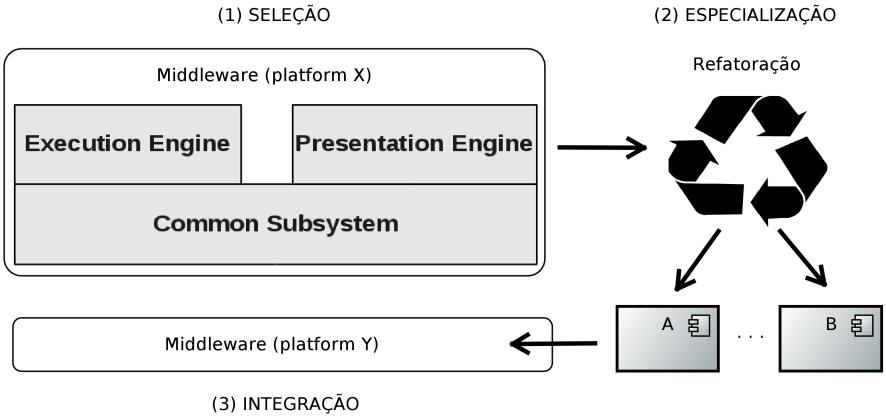


Figura 4.2: Modelo de reúso para um middleware para TVDi

1. Análise e escolha (seleção) das partes dos subsistemas declarativo e imperativo candidatas para o reúso. O subsistema comum — em tese, de reúso mais difícil — deve ficar fora desse processo de seleção.

2. Refatoração (especialização) da implementação original de modo a criar componentes reutilizáveis.

3. Porte (integração) dos componentes para uma nova plataforma de hardware, de modo a ser integrado em uma nova implementação do middleware para essa plataforma.

Essas três atividades correspondem, respectivamente, às atividades de seleção, especialização e integração, que compõe a base de qualquer processo de reúso, conforme visto nas Seção 2.1.

### 4.3.2 Contexto de Porte

Conforme visto no Capítulo 2, uma questão fundamental em um processo de porte é a definição do contexto de porte (Seção 2.3.3). A fim de limitar o escopo da proposta e facilitar sua aplicabilidade, o subsistema a ser refatorado/portado deverá ser mantido no mesmo sistema operacional original.

Foi mostrado na Seção 4.2.2 que mesmo num contexto de mudança de hardware as características de construção de um middleware para TVDi favorecem a portabilidade. Foi visto também que esse aspecto torna-se ainda mais relevante se implementação for mantida no contexto do mesmo SO.

Portanto, na abordagem proposta, será acrescentado ao contexto de porte a condição de manutenção do mesmo SO. Isso deve contribuir para simplificar o processo de porte, uma vez que à parte outras dependências externas, todo o código fonte pode ser reutilizado, já que todos os serviços e funções de SO utilizados serão mantidos, bem como suas respectivas assinaturas no código.

### 4.3.3 Foco em Portabilidade

O processo de refatoração a ser aplicado ao middleware deve ter dois objetivos principais: (i) Adaptar e especializar partes de um subsistema que se deseja reutilizar, separando-as do restante do middleware, de forma a transformá-las em componentes reutilizáveis; e (ii) Tornar esses componentes mais portáveis, quando comparados ao subsistema original, dentro do contexto de porte estabelecido.

Os dois objetivos estão relacionados, uma vez que para se atingir a ambos é necessário a refatoração das interfaces externas do subsistema escolhido. Transformações devem ser aplicadas ao middleware para se destacar os componentes do restante da implementação, criando interfaces bem definidas (Figura 4.2), conforme recomendado para um componente de software (CLEMENTS; NORTHROP, 2002)(KRUEGER, 1992).

Provavelmente, grande parte das transformações que serão efetuadas concentram-se nas fronteiras entre o subsistema a ser componentizado e o subsistema comum. Desse modo, a identificação das interfaces entre esses subsistemas constitui um aspecto chave desse processo.

Além das interfaces com o subsistema comum, é necessário identificar também as interfaces com outros possíveis componentes de software utilizados na implementação. Conforme visto anteriormente, a dependência externa de outro software constitui um dos fatores que podem

limitar bastante a portabilidade de um middleware para TVDi para outras plataformas.

Conforme a análise realizada na Seção 4.2, espera-se que uma parte significativa das dependências específicas de uma plataforma sejam eliminadas no processo de componentização, uma vez que os componentes criados serão separados do restante do middleware, em particular do subsistema comum. Isso deve contribuir de forma significativa para tornar os componentes mais portáveis quando comparado com o middleware original.

Quando se considera o processo de refatoração, uma questão importante diz respeito a quais transformações poderiam ser utilizadas no processo de componentização. Obviamente, isso depende em parte da implementação do middleware. Entretanto, é possível compor uma lista inicial com algumas refatorações e padrões de projeto (Seção 2.2) que podem ser úteis.

As refatorações devem criar interfaces externas bem definidas para o componente, o que contribui positivamente para reusabilidade e portabilidade. Transformações e padrões de projeto que introduzem abstrações e encapsulamento para subsistemas, módulos, classes, objetos, etc, são potenciais candidatos para essa tarefa (SCHMIDT; BUSCHMANN, 2003)(DEMEYER; DUCASSE; NIERSTRASZ, 2003)(FOWLER; BECK, 1999)(GAMMA et al., 1995) (KATAOKA et al., 2002)(KATAOKA, 2006). A Tabela 4.1 mostra uma lista com algumas dessas transformações.

Tabela 4.1: Refatorações e padrões de projeto para construção de interfaces

| **Tipo** | **Catálogo** | **Transformações** |
|---|---|---|
| Refatorações | (FOWLER; BECK, 1999) | Collapse Hierarchy, Extract Subclass, Extract Superclass, Extract Interface, Move Class, Remove Class, Pull Up Field, Pull Up Method, Push Down Field, Push Down Method |
| | (KERIEVSKY, 2005) | Extract Adapter, Unify Interfaces, Unify Interfaces with Adapter |
| Padrões de projeto | (GAMMA et al., 1995) | Adapter, Bridge, Facade, Proxy |

Algumas das transformações listadas foram introduzidas na Seção 2.2. De forma geral, todas podem ser encontradas nos catálogos de padrões de projetos e refatorações vistos nas Seções 2.2.3 e 2.2.4.

# 5 ESTUDO DE CASO: GINGA

## 5.1 Resumo dos Experimentos

Este capítulo apresenta uma aplicação da abordagem de reúso proposta no capítulo anterior em uma implementação do Ginga — o middleware do sistema brasileiro de TV Digital (Seção 3.2.4). Os experimentos compreenderam duas etapas principais:

1. Separação do subsistema Ginga-NCL do restante do middleware (e das dependências do Ginga-CC). O objetivo foi criar um componente de software reutilizável e mais portável dentro do contexto de desenvolvimento de novas versões do Ginga para plataformas de hardware diferentes da original, mas que utilizassem o mesmo sistema operacional alvo da implementação.

2. Utilização do novo Ginga-NCL no desenvolvimento de duas novas versões de Ginga para duas plataformas embarcadas de TVDi diferentes da original e diferentes entre si. Isso foi feito através da integração do Ginga-NCL com duas implementações diferentes de Ginga-CC (uma para cada plataforma de hardware).

## 5.2 Ginga para Linux/PC-x86

### 5.2.1 Descrição e Dependências

A implementação do Ginga escolhida é composta dos subsistemas Ginga-NCL e Ginga-CC, cujo código fonte está disponível em `http://www.softwarepublico.gov.br/ver-comunidade?community_id=1101545`.

O middleware foi desenvolvido em C++/C, para sistema operacional Linux, plataforma de hardware PC/x86 (e compatíveis) e deve ser compilado utilizando-se a suíte GNU[1] formada

[1] http://gcc.gnu.org/

pelos compiladores C++/C e as ferramentas Make e Autotools. A versão utilizada foi a 0.11.2, que provê suporte aos seguintes tipos de mídias para uso nas aplicações NCL: vídeo, áudio, imagem, texto, XHTML, Lua e NCL.

O middleware possui uma extensa lista de dependências externas (aplicativos, pacotes e bibliotecas) que devem ser mapeadas para permitir a sua correta compilação e execução. A Tabela 5.1 sumariza as principais.

Tabela 5.1: Lista de dependências externas da implementação do Ginga para PC/x86

| **Dependências** | **Descrição** |
|---|---|
| Bibliotecas, pacotes, aplicativos | DirectFB, libjpeg, libpng, libz, Pthread, libssl, faad2, FFmpeg, xine, fusionsound, lua, luasocket, video4linux, Xerces-C++, ... |
| **Total** | $> 30$ |

Uma das dependências mais importantes é a o pacote DirectFB[2], que provê recursos básicos para criação e manipulação de janelas gráficas, interação com dispositivos de entrada (teclado, etc.) e aceleração gráfica. Em conjunto com bibliotecas específicas para manipulação e exibição de fontes, texto, imagens, vídeo e áudio (como por exemplo FFmpeg[3], xine[4] e video4linux[5]), o DirectFB provê suporte à exibição de uma série de formatos de mídia.

A grande maioria dessas dependências ocorre na implementação do Ginga-CC (a exceção é o pacote Xerces-C++). Entretanto, como os recursos do CC são utilizados pelo Ginga-NCL, esse acaba herdando indiretamente essas dependências. O middleware também faz uso de um *parser* de XML — utilizado na interpretação dos documentos NCL pertecentes às aplicações — provido pelo pacote Xerces-C++[6].

A implementação depende também de bibliotecas de sistema C/C++. Essas bibliotecas permitem à aplicações C/C++ acessarem serviços e funções do sistema operacional e outros recursos auxiliares. As versões utilizadas são a GNU Standard C++ Library[7] e a GNU C Library[8].

Existem ainda outras duas dependências importantes em nível de linguagem e de SO: uma implementação de STL (*Standard Template Library*) e da biblioteca Pthread[9], que implementa

[2]http://directfb.org/
[3]http://ffmpeg.org/
[4]http://www.xine-project.org/
[5]http://linuxtv.org/
[6]http://xerces.apache.org/xerces-c/
[7]http://gcc.gnu.org/libstdc++/
[8]http://www.gnu.org/s/libc/
[9]POSIX Threads: padrão IEEE 1003.1c – `http://standards.ieee.org`

o suporte a processos leves ou *threads*. Entretanto, essas comumente fazem parte das suítes de compiladores C++/C, estando assim normalmente disponíveis.

### 5.2.2 Código Fonte e Binários

A implementação do Ginga é bastante extensa e complexa, como era de se esperar para um middleware para TVDi. São aproximadamente 90.000 (noventa mil) linhas de código distribuídas em mais de 850 (oitocentos e cinquenta) arquivos. Desse total, aproximadamente metade do código corresponde ao Ginga-NCL e pouco mais da metade corresponde ao Ginga-CC.

O código fonte é organizado em diretórios/pacotes e é composto de diversos subprojetos. O processo de compilação é trabalhoso, principalmente em função do mapeamento das dependências externas vistas anteriormente e das interdependências entre alguns dos subprojetos. Os principais pacotes que compõem a implementação são:

- ginga-cc: composto de diversos subprojetos que implementam as funcionalidades do subsistema Ginga-CC, como por exemplo, os exibidores de mídias (ou *players*), que implementam o suporte aos diversos formatos de mídias que o middleware pode exibir.
- ncl30: implementa *parser* NCL e conversor que cria modelo de dados para representar as aplicações NCL em um formato adequado para interpretação e execução pelo Ginga-NCL.
- gingancl: implementa o subsistema declarativo Ginga-NCL. É composto de diversos módulos, com destaque para o conversor NCL (que utiliza o pacote ncl30), o escalonador responsável pela condução da apresentação das aplicações NCL e o gerenciador de exibidores de mídias (que utiliza o pacote ginga-cc).

Os arquivos binários para instalação do middleware gerados pelo processo de compilação são compostos por um executável principal (ginga) e por diversas bibliotecas dinâmicas que implementam as funcionalidades do Ginga-NCL e do Ginga-CC.

### 5.2.3 Compilação e Testes

Antes de ser refatorado, o middleware foi compilado e testado em uma plataforma Linux/PC-x86 compatível, dotada de processador Intel® de 1,66GHz; 2GB RAM e resolução gráfica de 1280x800 pixels. Os compiladores utilizados foram o *g++*/*gcc* versão 4.3.2.

Para a compilação do middleware foram necessários ajustes em alguns arquivos do código fonte (basicamente arquivos de configuração e Makefiles) que serão descritos na próxima seção por estarem diretamente relacionados com o processo de refatoração e porte do middleware para outras plataformas.

Uma vez compilado e instalado, o middleware foi testado utilizando-se uma suíte contendo diversas aplicações NCL e NCL/Lua (a Listgem 3.1 mostra uma dessas aplicações). Nos testes efetuados, o middleware apresentou comportamento satisfatório[10]em termos de corretude na interpretação e execução das aplicações.

## 5.3 Criação do Componente Ginga-NCL

### 5.3.1 Preparação do Código

Antes da refatoração do middleware foi detectado a necessidade de ajustes em dezenas de arquivos configure.in e Makefile.am que são utilizados pelo GNU Autotools para geração de Makefiles — que contém os comandos e regras de compilação e instalação do middleware. O Autotools automatiza a geração de Makefile a partir dessas regras e das configurações do ambiente, tais como localização e nomes de compiladores, bibliotecas, arquivos de cabeçalho (.h, .hpp), etc.

A utilização correta do Autotools facilita bastante o trabalho de compilação e instalação de um software em ambientes diferentes daquele onde foi originalmente desenvolvido (distribuições diferentes de Linux, compiladores diferentes, etc.) e também a geração de versões para outras plataformas de hardware (*cross-compiling*). Dependências externas, como bibliotecas e arquivos de cabeçalho de terceiros, devem ser explicitamente indicadas nos arquivos .in, .am.

O Autotools provê meios para que se parametrize a localização no sistema de arquivos de todas as dependências externas bem como a localização final dos binários gerados. A parametrização é feita através de variáveis que são acessíveis dentro dos arquivos .in, .am e que são resolvidas apenas no momento da geração dos Makefiles. A não utilização desse recurso e a referência a caminhos absolutos no sistema de arquivos impede a reutilização dos arquivos .in, .am. Infelizmente, essa implementação do Ginga tinha essa deficiência, o que

[10]Durante o desenvolvimento deste trabalho foram identificadas inconsistências entre a implementação do Ginga utilizada e algumas especificações da norma do SBTVD para o Ginga-NCL (ABNT, 2011). Algumas funcionalidades não puderam ser testadas da maneira como estão descritas na norma em função de não estarem implementadas (como por exemplo alguns módulos Lua e algumas propriedades de objetos de mídia) ou por terem sido implementadas de forma diferente, o que requereu adaptações posteriores no código fonte.

motivou as modificações citadas.

Devido à grande quantidade de arquivos a serem modificados (total de 18 arquivos .in e 150 arquivos .am) e para evitar possíveis erros de edição, foram desenvolvidos *scripts* para fazer os ajustes necessários de forma automática. Essa etapa de adaptação dos arquivos .in, .am pode ser considerada um passo preliminar no processo de refatoração do middleware e foi fundamental para permitir seu porte para outras plataformas de hardware.

### 5.3.2 Análise do Código

A primeira atividade do processo de refatoração foi a identificação no código fonte das interfaces entre os subsistemas Ginga-NCL e Ginga-CC. Uma vez conhecida a forma como os subsistemas estão conectados, pode-se avaliar quais tranformações devem ser aplicadas para separar o Ginga-NCL do restante do middleware.

#### Classes do Ginga-NCL

A Figura 5.1 mostra um diagrama UML de classes simplificado do Ginga com algumas das principais classes do Ginga-NCL e seus relacionamentos com classes do Ginga-CC.

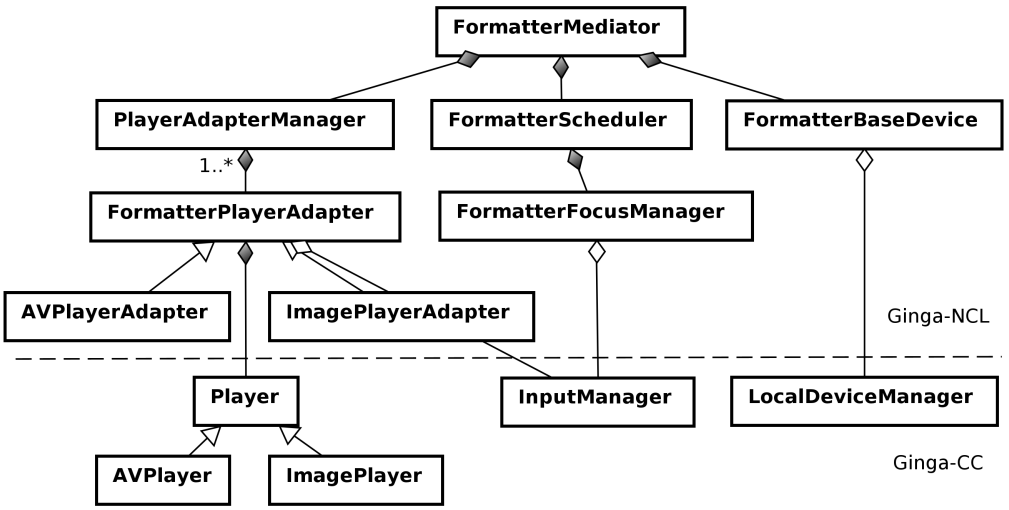


Figura 5.1: Classes do Ginga-NCL original e suas dependências de classes do Ginga-CC

A classe principal do Ginga-NCL é `FormatterMediator`, que recebe uma aplicação NCL a ser executada e é responsável por coordenar as ações entre as outras classes que implementam o subsistema declarativo.

A classe `FormatterConverter` é responsável por interpretar os documentos NCL que compõe a aplicação e converter as abstrações de alto nível da linguagem NCL em

um modelo de dados adequado para utilização pelo escalonador implementado pela classe `FormatterScheduler`).

A classe `FormatterScheduler` implementa o escalonador responsável por conduzir a execução das aplicações NCL a partir do modelo de dados criado por `FormatterConverter`. Além disso, agrega a classe `FormatterFocusManager`, responsável pelo controle de foco e navegação nas aplicações.

A classe `PlayerAdapterManager` é responsável por gerenciar as classes de adaptadores para *players* de mídias do Ginga-CC. Os adaptadores formam uma hierarquia de classes que implementam o padrão de projeto Adapter (Seção 2.2.3), cuja classe base é `FormatterPlayerAdapter`, conforme mostrado na Figura 5.2.

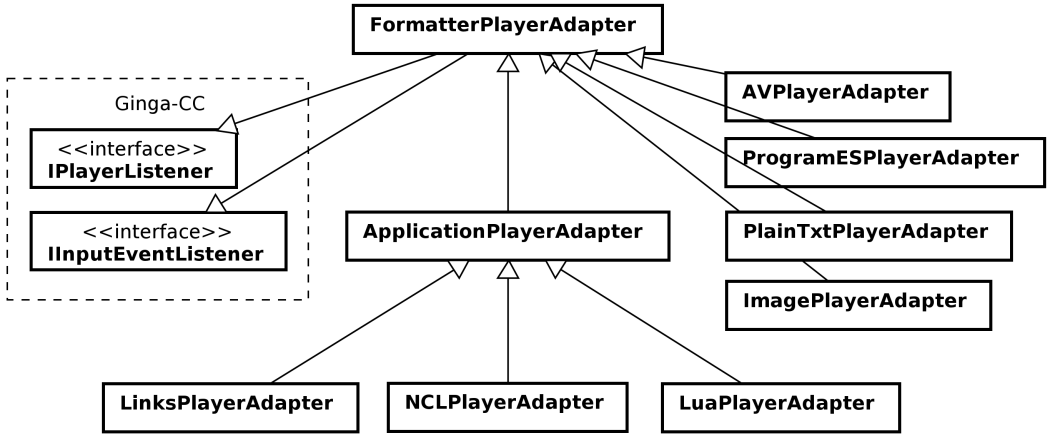


Figura 5.2: Classes de adaptadores para exibidores de mídias

A classe `FormatterBaseDevice` é responsável pelo controle das regiões/leiautes gráficos das aplicações NCL. Essa classe contém informações sobre as características físicas do dispositivo em que o middleware está sendo executado, tais como sua resolução gráfica.

A classe `PrivateBaseManager` é responsável por armazenar referências e gerenciar o acesso às aplicações NCL. Essa classe será melhor detalhada mais à frente, durante a descrição da refatoração do middleware.

Como mostrado nas Figuras 5.1 e 5.2, existem referências diretas a classes do Ginga-CC feitas a partir de classes do Ginga-NCL. Essas referências criam um acoplamento entre os subsistemas, o que dificulta sua separação e reutilização.

Algumas das principais interfaces identificadas entre o Ginga-NCL e o Ginga-CC serão pormenorizadas a seguir. Essas interfaces constituem basicamente dois tipos de classes: classes *abstratas* (identificadas com o estereótipo ou símbolo `<<interface>>` em todos os diagramas UML deste trabalho) e classes convencionais (ou *concretas*).

### Classes do Ginga-CC

O recurso de apresentação de mídias é disponibilizado através da classe `Player` e suas derivadas (Figura 5.3), tais como `AVPlayer` (áudio/vídeo) e `ImagePlayer` (imagens). As classes de exibidores são integradas ao Ginga-NCL através das classes de adaptadores (Figura 5.1).

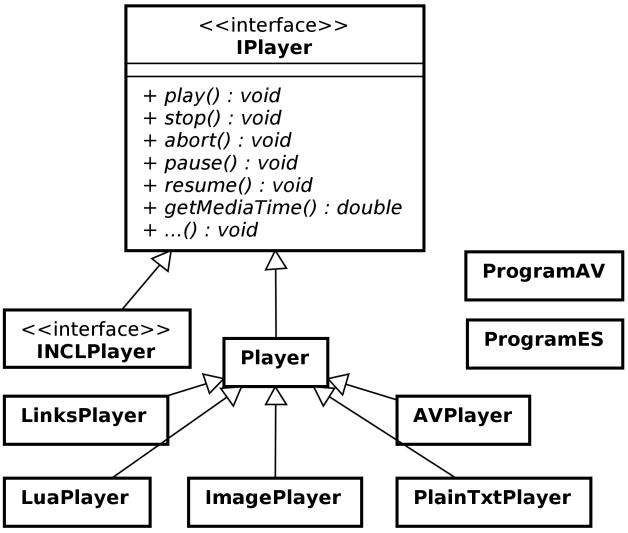


Figura 5.3: Classes de exibidores de mídias

A classe `Player` implementa a interface `IPlayer` (Listagem 5.1), que especifica o comportamento de um *player* de mídia genérico, definindo métodos para iniciar, pausar, parar (`play()`, `pause()`, `stop()`, entre outros) a apresentação de qualquer tipo de mídia.

Listing 5.1: Classe `IPlayer`

```
1  class IPlayer {
2        public:
3        virtual void play()=0;
4        virtual void stop()=0;
5        virtual void abort()=0;
6        virtual void pause()=0;
7        virtual void resume()=0;
8        virtual double getMediaTime()=0;
9        virtual string getPropertyValue(string name)=0;
10       virtual void setPropertyValue(string name, string value, ...)=0;
11       ...
12 };
```

Cada exibidor é responsável por fornecer suporte à apresentação de um determinado tipo de mídia e sua implementação é extremamente dependente dos recursos do pacote DirectFB e de outras bibliotecas gráficas, conforme visto na Seção 5.2.1.

As classes `ProgramAV` e `ProgramES` são tipos específicos de *players* que não fazem parte da hierarquia dos exibidores. Essas classes são utilizadas para dar suporte a um tipo especial de mídia (`sbtvd-ts://`) que representa um programa de TV dentro de uma aplicação NCL.

A classe `INCLPlayer` define a interface de um exibidor de mídia do tipo NCL. Um *player* de aplicação NCL é conhecido como Formatador e é implementado pela classe do Ginga-NCL `FormatterMediator`, apresentada anteriormente.

Para cada tipo de *player* do Ginga-CC existe uma classe `Adapter` correspondente no Ginga-NCL, cuja função seria encapsular/adequar a interface de um objeto do tipo `Player` ao subsistema declarativo. Isso constitui um dos pontos de integração entre esses subsistemas que foi explorado durante a refatoração do middleware.

As classes `Adapter` implementam também as interfaces `IPlayerListener` e `IInputEventListener` (Figura 5.2). A função da primeira é tornar os adaptadores ouvintes de eventos disparados pelos *players* (apresentação de mídias, mudanças de propriedades, etc.), enquanto a segunda permite o recebimento de eventos de entrada de dados do usuário.

A interação com o usuário nas aplicações é feita através do mecanismo de entrada de dados (via teclado, controle remoto, etc.) disponibilizado pela classe `InputManager`, cujos recursos são utilizados pelas classes de adaptadores/exibidores e pela classe `FormatterFocusManager` (Figura 5.1). Da mesma forma que as classes de exibidores, a implementação da classe `InputManager` também é bastante dependente do DirectFB.

Conforme visto anteriormente, o controle das regiões/leiautes gráficos das aplicações NCL é realizado a partir das informações da classe `FormatterBaseDevice`. Essas informações são obtidas da classe `LocalDeviceManager` (Figura 5.1), que implementa o acesso a recursos dos dispositivo de exibição da plataforma em que o middleware está sendo executado. Sabendo, por exemplo, a largura e a altura em pixels de uma tela, o gerenciador de leiautes pode organizar e controlar as regiões gráficas utilizadas pelas aplicações.

### 5.3.3 Refatoração do Middleware

Uma vez identificadas todas as dependências do Ginga-CC no susbsistema declarativo, iniciou-se o processo de refatoração do Ginga. Por uma questão de espaço, serão mostradas apenas as refatorações das interfaces utilizadas para acesso aos exibidores de mídias, entrada de dados e controle de leiautes gráficos, uma vez que essas cobrem boa parte das funcionalidades do middleware e ilustram bem os procedimentos realizados.

As descrições são feitas de forma sintética, de modo a simplificar o relato e facilitar o entendimento. Todas as transformações utilizadas e citadas foram referenciadas no Capítulo 4 (Tabela 4.1).

**Refatoração dos Adaptadores**

A análise da implementação da hierarquia de classes `Adapter` do Ginga (Figura 5.2) mostrou que essas possuiam poucas diferenças entre si. Na verdade, esse fato é um reflexo das semelhanças existentes entre as classes `Player`. Dessa forma, é natural que os adaptadores — cujas operações são baseadas nas interfaces das classes `IPlayer` e `Player` — sejam bastante semelhantes. As principais transformações aplicadas foram:

1. Refatoração Collapse Hierarchy para reduzir a hierarquia apenas à classe `FormatterPlayerAdapter`. Todas as demais classes `Adapter` foram removidas (refatoração Remove Class) e, juntamente com elas, as dependências das classes de exibidores do Ginga-CC (com exceção de `IPlayer` e `Player`). Essa refatoração também simplificou o código do gerenciador `PlayerAdapterManager`, que antes precisava instanciar e manipular adaptadores específicos baseado no tipo de mídia.

2. Refatoração Unify Interfaces with Adapter para eliminar referências diretas às classes de exibidores em qualquer ponto do código exceto dentro das classes `Adapter`. Os adaptadores possuiam métodos para retornar referências (ponteiros) para objetos `Player`, que por sua vez eram utilizados em outras classes. Isso criou uma forma mista de utilização dos recursos dos exibidores — ora sendo acessados através das classes `Adapter` ora de forma direta através da interface `IPlayer`, o que conflita diretamente com o conceito e intenção de utilização do padrão de projeto Adapter. Dessa forma, o resultado final da refatoração foi a utilização dos recursos apenas através da classe `FormatterPlayerAdapter` — que tornou-se, de fato, uma implementação do padrão Adapter.

3. Refatoração Extract Adapter para criar a subclasse `AppPlayerAdapter`. Devido a detalhes específicos da implementação do Ginga-NCL, foi visto posteriormente a necessidade de se criar um novo tipo de adaptador para interagir com exibidores de mídias do tipo NCL — Formatador NCL implementado pela classe `FormatterMediator`.

4. Refatoração Move Class para mover as classes `INCLPlayer`, `IPlayer`, `Player` e `IPlayerListener` do Ginga-CC para o Ginga-NCL. A classe `Player` implementa

o comportamento genérico de um exibidor de mídia e sua implementação é livre de qualquer dependência externa específica.

5. Refatoração Unify Interfaces para incorporar na classe `IPlayer` métodos públicos definidos apenas na subclasse `Player` e que eram necessários para compor uma interface genérica mais completa para os exibidores.

6. Refatoração Unify Interfaces para incorporar na classe `FormatterMediator` métodos herdados originalmente da classe `Player`, da qual foi disassociada. A classe `FormatterMediator` herdava de forma indireta a classe `IPlayer` duas vezes: através da classe `INCLPlayer` e também da classe `Player`. Apesar do comportamento semelhante ao de um exibidor genérico, foi mais conveniente disassociar a classe `FormatterMediator` da implementação da classe `Player` (outra motivação para isso foi a posterior refatoração da classe `Player` para criar a classe `PlayerProxy`, como será visto mais à frente).

É importante citar que a classe `IInputEventListener` que originalmente pertencia ao Ginga-CC também foi movida para o Ginga-NCL. A Figura 5.4 mostra a nova hierarquia de classes `Adapter` após a refatoração.

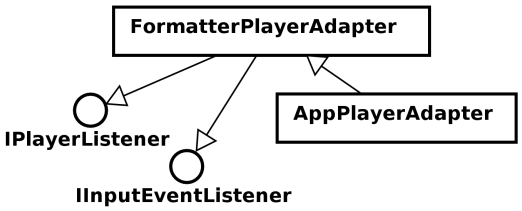


Figura 5.4: Classes de adaptadores após a refatoração

### Refatoração dos Exibidores

Na etapa de refatoração anterior, as classes `IPlayer` e `Player` foram movidas do Ginga-CC para dentro do Ginga-NCL. Esse foi um passo importante para eliminar do Ginga-NCL parte das dependências do Ginga-CC.

Foi visto também que a classe `IPlayer` especifica o comportamento básico de um exibidor. Essa API comum para os exibidores permitiu a construção de adaptadores mais genéricos. Isso contribuiu de forma decisiva para a criação de interfaces externas mais simples para o Ginga-NCL. Contudo, fazia-se necessário ainda prover meios para que os adaptadores pudessem ser integrados com os recursos de outras implementações de Ginga-CC.

Uma possível solução para isso seria a disponibilização por parte de um Ginga-CC de uma API que implementasse a interface `IPlayer`. Essa API poderia, por exemplo, retornar um ponteiro para um objeto do tipo `IPlayer`, que seria então utilizado pela classe `FormatterPlayerAdapter`.

Outra abordagem possível é a utilização na classe `FormatterPlayerAdapter` de um objeto que implemente o padrão de projeto Proxy (Seção 2.2.3) para encapsular e integrar os recursos de uma implementação qualquer de exibidor.

A inclusão de classes `Proxy` no lugar dos exibidores permite flexibilizar a interface externa para um hipotético Ginga-CC, que não precisa ter necessariamente qualquer ligação com a classe `IPlayer`. Por essa razão, essa foi a estratégia de refatoração escolhida nessa etapa.

A classe `Player` foi transformada em uma nova classe, chamada `PlayerProxy`, através de uma série de passos de refatoração, que incluiram modificações em atributos, métodos, etc. Todos os métodos públicos da nova classe são métodos vazios — que serão preenchidos de acordo com a interface disponibilizada por uma implementação de Ginga-CC. A Figura 5.5 mostra o resultado final das refatorações, com as interfaces criadas para exibidores externos.

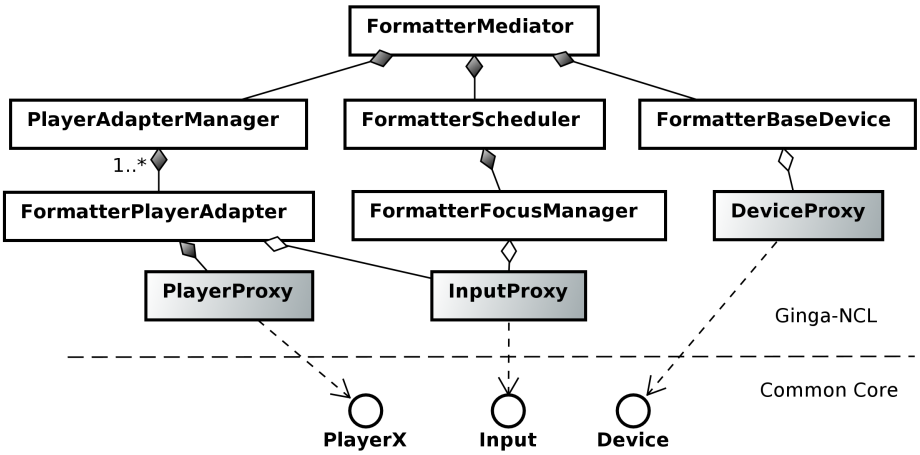


Figura 5.5: Ginga-NCL como um componente e sua integração com um Common Core através de suas interfaces externas

Uma implementação de Ginga-CC pode identificar o tipo de exibidor requerido pelo `PlayerProxy` através de dois atributos de mídia: o tipo MIME[11] e/ou o URI[12] (a exemplo dos uilizados para descrever endereços de recursos numa rede ou na internet). Os tipos URI e MIME utilizados pelo Ginga podem ser consultados em (ABNT, 2011)(SOARES; BARBOSA, 2009).

[11] http://www.iana.org/assingments/media-types/index.html
[12] http://www.w3.org/Addressing/URL/Overview.html

### Entrada de Dados e Controle de Leiautes

Para eliminar a dependência das classes `InputManager` e `LocalDeviceManager`, a estratégia adotada foi semelhante à utilizada na refatoração das interfaces com os exibidores de mídias. Assim, essas classes foram refatoradas e deram lugar, respectivamente, às classes `InputProxy` e `DeviceProxy` (que foram posteriormente incluídas no Ginga-NCL). A Figura 5.5 mostra o resultado final dessas refatorações.

### Outras Refatorações

O Ginga-NCL possui outras dependências do Ginga-CC que não foram refatoradas em função de limitações de tempo para essa pesquisa. Entretanto, acredita-se que os princípios utilizados nas refatorações descritas anteriormente podem ser aplicados igualmente a essas.

Uma sugestão seria o recurso conhecido como Comandos de Edição (ABNT, 2011)(SOARES; BARBOSA, 2009), utilizado pelas classes `PrivateBaseManager` e `FormatterMediator` do Ginga-NCL. Trata-se de mensagens remotas recebidas pelo middleware e que são utilizadas para controlar a apresentação e fazer a edição de documentos NCL durante a execução de uma aplicação.

Os comandos de edição podem ser recebidos basicamente de duas formas: (i) através do Carrossel de Dados (mecanismo utilizado para transporte das aplicações, visto na Seção 3.1); e (ii) através do Canal de Retorno (ver Seção 3.1). Nesse caso, os comandos poderiam ser repassados ao Ginga-NCL através de um Proxy, semelhante aos vistos anteriormente.

### Considerações sobre as Refatorações

As refatorações descritas para as interfaces de exibidores de mídias, entrada de dados e controle de regiões gráficas cobriram uma parte substancial das interfaces originais existentes entre o Ginga-NCL e o Ginga-CC. A Figura 5.5 mostra o resultado das refatorações e as interfaces resultantes do componente Ginga-NCL.

As interfaces refatoradas são suficientes para se testar um subconjunto bastante significativo das funcionalidades do middleware. As interfaces que não foram refatoradas — tais como as utilizadas para os comandos de edição — foram desativadas.

É importante ressaltar que em função da proposta do trabalho, o foco das refatorações foram as interfaces entre os subsistemas Ginga-NCL e Ginga-CC. Entretanto, outras oportunidades de refatoração não diretamente relacionadas ao objetivo do trabalho foram identificadas.

Houve também outras refatorações efetuadas como consequência da separação dos subsistemas. Suas descrições foram omitidas em favor da simplificação do relato e para dar ênfase diretamente às transformações descritas.

Um resultado importante da refatoração dos exibidores de mídias é que a forma de utilização das classes `FormatterPlayerAdapter` e `PlayerProxy` é genérica o suficiente para suportar qualquer tipo de mídia. De forma geral, a introdução das classes Proxy promoveu os seguintes aspectos importantes na implementação:

- Possibilitar o teste do Ginga-NCL — ainda que de forma limitada — sem a necessidade de uma implementação de Ginga-CC (os objetos Proxy podem emular os dispositivos, mesmo que esses não estejam disponíveis).
- Controlar/proteger/limitar o acesso aos recursos da plataforma, que é um aspecto fundamental na implementação de middleware para TVDi, conforme visto na Seção 3.2.3 (essa funcionalidade é o que Gamma et al. (GAMMA et al., 1995) define como *protection proxy*).
- Eliminar completamente quaisquer referências aos exibidores de mídias de outras partes do Ginga-NCL que não as classes `Adapter`, permitindo que o restante do subsistema evolua de maneira independente.
- Delimitar a fronteira de integração do componente Ginga-NCL com outras implementações de Ginga-CC às classes `PlayerProxy`, `InputProxy` e `DeviceProxy`.

### 5.3.4 Avaliação do Grau de Portabilidade

A separação do subsistema Ginga-NCL do restante da implementação original do Ginga teve dois objetivos: (i) criar um componente que pudesse ser reutilizado na implementação de novas versões do Ginga para outras plataformas de hardware; e (ii) como consequencia de (i), eliminar as dependências específicas da implementação para PC/x86, de modo a se obter um componente mais portável.

O componente Ginga-NCL criado possui muito menos dependências externas do que sua versão da implementação para PC/x86. A Tabela 5.2 sumariza as dependências que restaram da implementação original.

Comparando com a lista de dependências original (Tabela 5.1), nota-se a significativa redução da quantidade de dependências, de mais de 30 (trinta) para apenas 2 (duas). Isso era

Tabela 5.2: Lista de dependências externas do componente Ginga-NCL

| **Dependências** | **Descrição** |
|---|---|
| Bibliotecas, pacotes | Pthread, Xerces-C++ |
| **Total** | 2 |

esperado, uma vez que a maioria das dependências eram utilizadas na implementação do Ginga-CC e, consequentemente, uma vez eliminada do Ginga-NCL a dependência desse subsistema, o número de dependências externas para o Ginga-NCL deveria ser reduzido.

Assim, conforme visto na Seção 2.3.4, em função da diminuição da quantidade de dependências, pode-se dizer que o componente Ginga-NCL possui um grau de portabilidade maior do que o subsistema Ginga-NCL original.

Na Seção 2.1 foi visto as métricas DIT (*Depth of Inheritance Tree*), NOC (*Number of Children*) e CBO (*Coupling Between Object*) e como essas podem ser utilizadas para calcular o valor da reusabilidade R (Equação 2.1) para uma classe e/ou diagrama de classes.

A Tabela 5.3 mostra os valores de DIT, NOC, CBO e R calculados para algumas das classes mais importantes do subsistema Ginga-NCL original (inclusive a classe principal, `FormatterMediator`).

Tabela 5.3: Métricas para as classes do Ginga-NCL original

| **Classe** | **DIT** | **NOC** | **CBO** | **R** | **Rn** |
|---|---|---|---|---|---|
| `FormatterMediator` | 1 | 0 | 31 | -14,5 | 0,5 |
| `FormatterPlayerAdapter` | 1 | 9 | 32 | -6 | 9 |
| `FormatterConverter` | 1 | 0 | 24 | -11 | 4 |
| `FormatterEvent` | 1 | 3 | 25 | -8,5 | 6,5 |
| `FormatterFocusManager` | 1 | 0 | 15 | -6,5 | 8,5 |
| `FormatterScheduler` | 1 | 0 | 27 | -12,5 | 2,5 |
| `PlayerAdapterManager` | 1 | 0 | 17 | -7,5 | 7,5 |
| `CascadingDescriptor` | 0 | 0 | 18 | -9 | 6 |
| `CompositeExecutionObject` | 1 | 1 | 21 | -8,5 | 6,5 |
| `ExecutionObject` | 0 | 2 | 30 | -13 | 2 |

Por conveniência, os valores de R na Tabela 5.3 foram normalizados para números positivos (somando-se o valor 15). Os valores normalizados de R são mostrados na coluna Rn.

A Tabela 5.4 mostra os valores de DIT, NOC, CBO e R calculados para as mesmas classes

da Tabela 5.3, mas agora para o componente Ginga-NCL. O mesmo critério de normalização anterior foi utilizado para gerar os valores da coluna Rn.

Tabela 5.4: Métricas para as classes do componente Ginga-NCL

| **Classe** | **DIT** | **NOC** | **CBO** | **R** | **Rn** |
|---|---|---|---|---|---|
| `FormatterMediator` | 0 | 0 | 16 | -8 | 7 |
| `FormatterPlayerAdapter` | 1 | 1 | 13 | -4,5 | 10,5 |
| `FormatterConverter` | 1 | 0 | 20 | -9 | 6 |
| `FormatterEvent` | 1 | 3 | 20 | -6 | 9 |
| `FormatterFocusManager` | 1 | 0 | 9 | -3,5 | 11,5 |
| `FormatterScheduler` | 1 | 0 | 21 | -9,5 | 5,5 |
| `PlayerAdapterManager` | 1 | 0 | 11 | -4,5 | 10,5 |
| `CascadingDescriptor` | 0 | 0 | 12 | -6 | 9 |
| `CompositeExecutionObject` | 1 | 1 | 15 | -5,5 | 9,5 |
| `ExecutionObject` | 0 | 2 | 25 | -10,5 | 4,5 |

Comparando-se as Tabelas 5.3 e 5.4, nota-se que os valores de DIT e NOC praticamente não mudaram — com exceção para a classe `FormatterPlayerAdapter`. Entretanto, houve redução significativa dos valores de CBO, que foram refletidos nos valores de R e Rn mostrados. De forma geral, o componente Ginga-NCL apresenta valores de R maiores do que os do Ginga-NCL original, o que indica que o componente possui melhor capacidade de reutilização do que este.

Essa melhora é um reflexo direto da refatoração realizada no Ginga-NCL, que eliminou todas as dependências externas de classes do Ginga-CC. Os valores maiores de R sugerem melhorias em aspectos como modularidade e reusabilidade (CHIDAMBER; KEMERER, 1994)(BHATIA; MANN, 2008) e a melhora do grau de portabilidade também reforça essa conclusão, uma vez que porte é uma forma de reúso.

É importante comentar sobre a classe `FormatterPlayerAdapter`, cuja hierarquia de classes `Adapter` (Figura 5.2) passou por transformações significativas. Deve-se lembrar que a refatoração reduziu o número dessas subclasses, cujo efeito pode ser percebido na diminuição do NOC — o que, a princípio, tende a diminuir a reusabilidade. Entretanto, a diminuição do NOC foi compensada pela redução do acoplamento a outras classes, evidenciada pelo valor menor de CBO, o que contribuiu para melhorar a reusabilidade R.

Portanto, as métricas de portabilidade e reusabilidade apresentadas indicam que, dentro do contexto de porte proposto, a versão componentizada do Ginga-NCL possui um maior grau de portabilidade e maior capacidade de reúso do que sua versão original. Isso pôde ser verificado na prática, através dos experimentos de porte e reúso descritos a seguir.

## 5.4 Reutilização do Componente Ginga-NCL

### 5.4.1 Plataformas e Porte

Para avaliar o esforço de porte da versão componentizada do Ginga-NCL para outras plataformas de hardware, foram utilizadas duas plataformas embarcadas para TVDi, que são mostradas na Tabela 5.5. Plataformas similares podem ser encontradas nos sítios de internet dos respectivos fabricantes (ST[13] e Broadcom[14]).

Tabela 5.5: Plataformas de TVDi utilizadas nos experimentos

| Hardware | Descrição |
|---|---|
| S | ST® – cpu SH4 32-bits@266MHz; processadores para áudio/vídeo/gráficos; 256MB RAM; 1920x1080 pixels; SO Linux; compiladores sh4-linux-gcc/g++ versões 4.1.1. |
| M | Broadcom® – cpu MIPS 32-bits@300MHz; processadores para áudio/vídeo/gráficos; 512MB RAM; 1920x1080 pixels; SO Linux; compiladores mips-linux-gcc/g++ versões 4.2.3. |

Antes do porte para as plataformas, o componente Ginga-NCL foi compilado novamente para PC/x86. Como era esperado, o processo foi bem mais simples e rápido do que para o middleware original, em função da quantidade reduzida de dependências externas (Pthread e Xerces-C++). Na versão original, para que o Ginga-NCL seja compilado, é necessário antes compilar e instalar todo o subsistema Ginga-CC (que é composto de vários módulos) e suas dependências específicas.

Para a compilação do Ginga-NCL para as plataformas, foi necessário apenas obter para cada uma delas as versões específicas das biblitotecas Pthread e Xerces-C++ — a primeira normalmente é disponibilizada juntamente com os compiladores C/C++, enquanto a segunda possui código fonte disponível, cujo porte para qualquer ambiente baseado em UNIX resume-se apenas ao processo de compilação.

A maior simplicidade dessa etapa preliminar de porte serve como mais uma confirmação de que o produto final do processo de refatoração do Ginga é um componente Ginga-NCL

[13] http://www.st.com
[14] http://www.broadcom.com/

mais portável do que sua versão original para PC/x86. A próxima seção descreve como esse componente foi reutilizado na implementação de duas novas instâncias do Ginga.

### 5.4.2 Versões de Middleware Construídas

O componente Ginga-NCL foi utilizado na construção de duas novas versões do Ginga, uma para cada plataforma de hardware da Tabela 5.5. No desenvolvimento das versões de middleware foram utilizadas duas versões específicas de Common Core (uma para cada plataforma), mostradas na Tabela 5.6.

Tabela 5.6: Implementações de Ginga-CC utilizadas nos experimentos

| Ginga-CC | Hardware | Descrição |
|---|---|---|
| CC1 | S | Suporte a mídias do tipo imagem e texto; controle remoto; leiautes gráficos. |
| CC2 | M | Suporte a mídias do tipo imagem, texto, programas de TV e Lua; controle remoto; leiautes gráficos. |

A implementação do Common Core CC1 foi fornecida em código fonte, enquanto que o CC2 foi fornecido apenas no formato binário/executável para sua respectiva plataforma. Uma particularidade do Common Core CC2 é que o mesmo funciona como uma aplicação autônoma, tendo seu próprio processo no SO Linux.

O componente Ginga-NCL foi integrado com as duas versões de Common Core para construir duas novas instâncias do Ginga, A e B, conforme mostrado na Tabela 5.7.

Tabela 5.7: Instâncias de Ginga construídas a partir do componente Ginga-NCL

| Ginga | Hardware | Descrição |
|---|---|---|
| A | S | Integração Ginga-NCL $\leftrightarrow$ CC1 |
| B | M | Integração Ginga-NCL $\leftrightarrow$ CC2 |

A integração entre o Ginga-NCL e os Common Core ocorreu através das classes `PlayerProxy`, `InputProxy` e `DeviceProxy`, conforme ilustrado na Figura 5.5. Ambos os Common Core disponibilizam APIs para acesso a recursos de exibidores de mídias, entrada de dados e controle de dispositivos de exibição.

Para a integração, foi necessário apenas ajustes em alguns Makefiles e arquivos fonte do Ginga-NCL para a introdução das dependências relativas aos Common Core CC1 e CC2. Feito isso, o processo de compilação e empacotamento de binários para as plataformas ocorreu sem problemas.

### 5.4.3 Testes

Os testes dos Ginga A e B foram efetuados utilizando-se a mesma suíte de testes usada para testar a versão original do Ginga para PC/x86. Contudo, nem todas as aplicações puderam ser testadas em ambos os middleware em função da limitação dos tipos de mídias suportadas pelo Common Core CC1 (e, consequentemente, pelo Ginga A).

De forma geral, as aplicações foram executadas corretamente quando comparadas com a versão original do middleware para PC/x86. Entretanto, alguns problemas foram detectados durante os testes.

Ao se monitorar a utilização de CPU e memória por parte dos processos dos middleware nas plataformas, foi notado um incremento constante no consumo de memória (mesmo com aplicações que não faziam nada), indicando problemas de vazamento de memória.

Os problemas de vazamento eram devidos à falta de liberação de recursos instanciados no Ginga-NCL (objetos criados com o operador `new` de C++). Na verdade, os problemas já existiam na implementação original e não haviam sido notados antes na plataforma PC/x86.

O uso de ferramentas de depuração como Valgrind[15] e GDB[16] foram fundamentais na identificação dos pontos problemáticos no código. Apesar da utilização do Valgrind ser inviável em plataformas com recursos computacionais limitados (como as utilizadas nos experimentos), foi possível fazer parte do processo de depuração em PC/x86.

Outro problema eventual detectado nos testes estava relacionado com questões de sincronização de eventos de apresentação e atribuição de propriedades para as mídias. Algumas vezes, mudanças no estado de uma mídia (por exemplo uma mídia em execução sendo interrompida) não eram detectadas no momento adequado pelo Ginga-NCL, o que fazia com que ações sincronizadas com essas mudanças não ocorressem.

A sincronização de eventos é implementada no Ginga através de recursos da biblioteca Pthread, tais como semáforos, mutexes, etc. Os problemas de sincronização encontrados foram atribuídos às diferenças de velocidade na execução dos middleware nas plataformas quando

[15] http://valgrind.org
[16] http://www.gnu.org/s/gdb

comparados com a versão em PC/x86.

Como os problemas de sincronização não eram reproduzidos em PC/x86, foi necessário depurar o software nas próprias plataformas. O hardware S (Tabela 5.5) foi escolhido inicialmente para fazer a depuração por apresentar uma versão já portada do GDB Server[17], que permite fazer a depuração remota da plataforma em PC/x86 via rede.

Com a utilização do GDB em PC/x86 e do GDB Server na plataforma, foi possível identificar os pontos do código fonte que eram problemáticos com relação a sincronização e implementar as correções necessárias.

## 5.5 Discussão dos Resultados

A aplicação da abordagem de reúso proposta no Capítulo 4 em uma implementação real de middleware para TVDi permitiu a geração de um componente Ginga-NCL portável e reutilizável dentro de um determinado contexto de porte/reúso.

Integração com uma implementação de Ginga-CC, compilação e testes constituem as atividades para a construção de uma nova versão de Ginga utilizando o componente construído. Esse ciclo pode ser repetido para inúmeras plataformas no futuro, desde que seja mantido o mesmo contexto de porte.

A reutilização do componente Ginga-NCL permite agilizar o desenvolvimento de novas instâncias do Ginga, uma vez que o esforço investido no desenvolvimento e testes do Ginga-NCL pode ser economizado em novas implementações do middleware.

É necessário lembrar, entretanto, que houve um custo inicial para o processo de refatoração da implementação do Ginga para PC/x86. Esse processo foi o fator determinante para a relativa facilidade de porte e reúso do componente Ginga-NCL em novas plataformas de hardware.

Apesar do esforço inicial investido no processo de refatoração, acredita-se que em função da complexidade e dimensões do middleware, esse esforço inicial foi compensado pelos resultados obtidos, uma vez que o componente Ginga-NCL foi utilizado como base para o desenvolvimento de uma versão do middleware embarcada em um produto real.

Algumas observações finais com relação aos experimentos realizados:

- A arquitetura da implementação do Ginga para PC/x86 mostrou-se de acordo com os padrões de construção de middleware para TVDi. Isso contribuiu para facilitar a

[17] http://www.stlinux.com/devel/debug/user/cross

separação do Ginga-NCL da implementação original e transformá-lo em um componente reutilizável.

- O processo de refatoração do middleware descrito neste trabalho durou cerca de 1 mês em uma fase preliminar (planejamento, análise, etc.) utilizando 2 desenvolvedores; depois mais 2 meses em uma segunda fase (refatorações e testes em PC/x86) com 3 desenvolvedores; a fase final (refatorações, integração com Common Core, testes, etc.) durou cerca de 3 meses, com também 3 desenvolvedores.

- Embora o contexto de porte proposto tenha escopo limitado, dentro desse contexto, o modelo de reúso certamente mostrou-se útil e propicia o desenvolvimento mais ágil de versões do Ginga para outras plataformas de hardware.

# 6 CONCLUSÕES

## 6.1 Contribuições

Esta dissertação apresentou uma abordagem para reúso de implementações de middleware para TVDi no desenvolvimento de novas instâncias desses middleware para plataformas de hardware diferentes da original.

Uma abordagem de refatoração com foco em portabilidade foi proposta para ser aplicada em implementações de middleware para TVDi que não tenham sido desenvolvidas com enfoque no reúso, de modo a adaptá-los para reúso/porte em outras plataformas de hardware.

Um modelo de reúso foi proposto e aplicado em uma implementação do middleware Ginga, que foi refatorado para criar um componente reutilizável/portável, que foi utilizado na construção de duas novas instâncias do Ginga para duas plataformas de TVDi diferentes.

Os experimentos mostraram que, ainda que o modelo proposto possua um contexto restrito de aplicabilidade, a abordagem pode contribuir para agilizar o desenvolvimento de novas instâncias de um middleware para TVDi para outras plataformas de hardware.

As principais contribuições deste trabalho são:

1. Estudo sobre as partes de uma implementação de middleware para TVDi que são mais adequadas para porte para outras plataformas de hardware.

2. As estratégias de refatoração e o modelo de reúso proposto para essas partes mais portáveis de um middleware para TVDi.

3. Mostrar que dentro de um contexto específico a abordagem de reúso proposta para middleware para TVDi pode ser viável na prática.

4. A experiência de refatoração de uma implementação real de um middleware para TV Digital e seu reúso no desenvolvimento de um novo produto.

5. Um componente de software (Ginga-NCL) reutilizável e portável, que pode ser utilizado no desenvolvimento de novas instâncias de um middleware para TVDi.

## 6.2 Limitações

A abordagem de reúso proposta tem como uma das suas premissas a manutenção da linguagem original no qual o middleware foi desenvolvido e do sistema operacional original, de forma a limitar o contexto de porte e facilitar a sua aplicação.

Apesar dessa restrição no contexto da solução, as experimentações mostraram que pode ser muito interessante a reutilização de partes de implementações existentes de um middleware para TVDi. O estudo de caso apresentado é uma indicação disso, uma vez que a versão componentizada do Ginga-NCL foi utilizada no desenvolvimento de um produto real.

## 6.3 Trabalhos Relacionados

A maior parte dos trabalhos consultados sobre desenvolvimento de middleware para TVDi tratam do desenvolvimento de soluções completas ou de frameworks:

Em (GAO; WANG; NI, 2009), uma arquitetura e projeto de middleware para TVDi reutilizável/portável é apresentado, mostrando seu porte para várias plataformas de hardware distintas. Ressalta-se, entretanto, o fato desse ser um trabalho cujo objetivo era implementar um middleware completo, projetado para reúso, ao contrário do trabalho apresentado neste artigo, que propõe a adaptação/reúso/porte de middleware já existente e não necessariamente voltados para o reúso.

Um framework para desenvolvimento de middleware para TVDi baseado em uma arquitetura de componentes é apresentado em (ELIAS et al., 2004) (cujo trabalho é complementado em (LOPES et al., 2006)). A solução permite o desenvolvimento de middleware baseado num modelo de componentes reutilizáveis e adaptáveis, que podem ser combinados e configurados em função das características de cada plataforma (hardware, SO, etc.), o que o torna bastante flexível. Entretanto, deve-se ressaltar que, de forma semelhante a (GAO; WANG; NI, 2009), trata-se de uma solução projetada com foco em reúso — diversamente da proposta deste trabalho, que é criar um componente reutilizável a partir de implementações existentes de middleware para TVDi.

Uma implementação do Ginga-NCL para utilização em dispositivos portáteis com sistema operacional Symbian é apresentada em (CRUZ; MORENO; SOARES, 2008). Esse trabalho é

baseado na implementação do Ginga para PC/x86, que foi adaptada/portada para Symbian, mas sem a criação de componentes reutilizáveis, que é a abordagem proposta nesta dissertação.

Alguns trabalhos utilizam refatoração para desenvolver abordagens para melhorar outras características não funcionais de um software, que não portabilidade:

Um estudo de caso de refatoração da versão do Ginga para PC/x86 é apresentado em (SARAIVA et al., 2010), que combina conceitos e técnicas de linha de produtos de software e programação orientada a aspectos para criar uma nova arquitetura mais modular e componentizada para o subsistema Ginga-CC.

Em (TAHVILDARI; KONTOGIANNIS, 2002) é proposta uma abordagem para reengenharia de sistemas orientados a objeto baseada em características de qualidade. Um caso voltado para manutenibilidade é mostrado.

Outro exemplo de aplicação de refatoração para melhorar a manutenibilidade é apresentado em (KATAOKA et al., 2002)(KATAOKA, 2006), onde são propostos também métodos para calcular o efeito das refatorações nessa característica específica.

## 6.4 Trabalhos Futuros

A abordagem de reúso proposta neste trabalho poderia ser estendida para incluir a análise do efeito das refatorações em outras características do middleware, como desempenho. Esse é um aspecto importante, principalmente para sistemas embarcados.

Um outro estudo que pode ser interessante é o da refatoração para portabilidade do subsistema comum de um middleware para TVDi. Entretanto, essa é uma questão que pode ser muito dependente de uma implementação e plataformas específicas, conforme visto neste trabalho.

Uma outra possibilidade de extensão da abordagem seria a expansão do contexto de porte, para incluir, por exemplo, o porte do middleware para um outro sistema operacional.

# REFERÊNCIAS BIBLIOGRÁFICAS

ABNT. *NBR 15606-2. Televisão digital terrestre - Codificação de dados e especificações de transmissão para radiodifusão digital Parte 2: Ginga-NCL para receptores fixos e móveis - Linguagem de aplicação XML para codificação de aplicações*. Rio de Janeiro, Brasil, 2011.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. 2. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321154959.

BHATIA, P. K.; MANN, R. An approach to measure software reusability of oo design. In: *Proceedings of 2nd National Conference on Challenges & Opportunities in Information Technology*. Mandi Gobindgarh, Punjab: RIMT-IET, 2008. (COIT-2008).

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 20, p. 476–493, June 1994. ISSN 0098-5589.

CLEMENTS, P.; NORTHROP, L. *Software Product Lines: Practices and Patterns*. Boston: Addison-Wesley, 2002.

CRINON, R. et al. Data broadcasting and interactive television. *Proceedings of the IEEE*, v. 94, n. 1, p. 102–118, Jan 2006. ISSN 0018-9219.

CRUZ, V. M.; MORENO, M. F.; SOARES, L. F. G. Ginga-ncl: implementaçao de referência para dispositivos portáteis. In: *WebMedia '08: Proceedings of the 14th Brazilian Symposium on Multimedia and the Web*. New York, NY, USA: ACM, 2008. p. 67–74. ISBN 978-1-60558-170-5.

DEMEYER, S.; DUCASSE, S.; NIERSTRASZ, O. *Object-oriented reengineering patterns*. Switzerland: Morgan Kaufman Publishers, 2003. (The Morgan Kaufmann Series in Software Engineering and Programming). ISBN 9781558606395.

ELIAS, G. et al. Exploring an open, distributed multimedia framework to design and develop an adaptive middleware for interactive digital television systems. In: *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2004. p. 1258–1264. ISBN 1-58113-812-1.

FERREIRA, G. D. et al. Ginga-ncl em dispositivos portáteis: Uma implementaçao para a plataforma android. In: *WebMedia 2010: Simpósio Brasileiro de Sistemas Multimidia e Sistemas Web*. Belo Horizonte, Brazil: Anais do WebMedia, 2010.

FILHO, G. L. de S.; LEITE, L. E. C.; BATISTA, C. E. C. F. Ginga-j: The procedural middleware for the brazilian digital tv system. *JBCS: Journal of the Brazilian Computer Society*, SBC - Sociedade Brasileira de Computação, v. 13, n. 1, p. 47–56, March 2007. ISSN 0104-6500.

FOWLER, M.; BECK, K. *Refactoring: improving the design of existing code*. Westford, Massachusetts: Addison-Wesley, 1999. (Addison-Wesley object technology series). ISBN 9780201485677.

FRAKES, W. B.; KANG, K. Software reuse research: Status and future. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 31, p. 529–536, July 2005. ISSN 0098-5589.

GAMMA, E. et al. *Design patterns: elements of reusable object-oriented software*. Westford, Massachusetts: Addison-Wesley, 1995. (Addison-Wesley professional computing series). ISBN 9780201633610.

GAO, C.; WANG, L.; NI, H. Software design methodology for set-top box. In: *ICIE '09: Proceedings of the 2009 WASE International Conference on Information Engineering*. Washington, DC, USA: IEEE Computer Society, 2009. p. 302–305. ISBN 978-0-7695-3679-8.

GARLAN, D.; SHAW, M. An introduction to software architecture. In: *Advances in Software Engineering and Knowledge Engineering*. New Jersey: World Scientific Publishing Company, 1993. p. 1–39.

HAKUTA, M.; OHMINAMI, M. A study of software portability evaluation. *Journal of Systems and Software*, v. 38, n. 2, p. 145 – 154, 1997. ISSN 0164-1212.

ISO/IEC. *13818-1. Information technology - Generic coding of moving pictures and associated audio information: Systems*. Geneva, Switzerland, 2000.

ITU-T. *J.200. Series J: Cable Networks and Transmission of Television, Sound Programme and Other Multimedia Signals - Application for Interactive Digital Television - Worldwide common core – Application environment for digital interactive television services*. Geneva, Switzerland, 2001.

ITU-T. *J.202. Series J: Cable Networks and Transmission of Television, Sound Programme and Other Multimedia Signals - Application for Interactive Digital Television - Harmonization of procedural content formats for interactive TV applications*. Geneva, Switzerland, 2003.

ITU-T. *J.201. Series J: Cable Networks and Transmission of Television, Sound Programme and Other Multimedia Signals - Application for Interactive Digital Television - Harmonization of declarative content format for interactive television applications*. Geneva, Switzerland, 2008.

KATAOKA, Y. *Toward Practical Application of Program Refactoring*. Tese (Doutorado), Osaka, Japan, 2006.

KATAOKA, Y. et al. A quantitative evaluation of maintainability enhancement by refactoring. In: *Proceedings of the International Conference on Software Maintenance (ICSM'02)*. Washington, DC, USA: IEEE Computer Society, 2002. p. 576–585. ISBN 0-7695-1819-2.

KERIEVSKY, J. *Refactoring to patterns*. Westford, Massachusetts: Addison-Wesley, 2005. (The Addison-Wesley signature series). ISBN 9780321213358.

KRUEGER, C. W. Software reuse. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 24, n. 2, p. 131–183, 1992. ISSN 0360-0300.

LOPES, A. et al. A meta-component model for dynamic adaptation support in a middleware system for interactive tv. In: *WebMedia '06: Proceedings of the 12th Brazilian Symposium on Multimedia and the web*. New York, NY, USA: ACM, 2006. p. 193–202. ISBN 85-7669-100-0.

MENS, T.; TOURWé, T. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 30, p. 126–139, February 2004. ISSN 0098-5589.

MOONEY, J. Developing portable software. In: REIS, R. (Ed.). *Information Technology*. Morgantown, USA: Springer Boston, 2004, (IFIP International Federation for Information Processing, v. 157). p. 55–84. ISBN 978-1-4020-8158-3.

MOONEY, J. D. Portability and reusability: common issues and differences. In: *Proceedings of the 1995 ACM 23rd annual conference on Computer science*. New York, NY, USA: ACM, 1995. (CSC '95), p. 150–156. ISBN 0-89791-737-5.

MORRIS, S.; CHAIGNEAU, A. *Interactive TV Standards*. Oxford: Focal Press, Elsevier, 2005. ISBN 9780240806662.

OPDYKE, W. F. *Refactoring object-oriented frameworks*. Tese (Doutorado), Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

PIESING, J. The dvb multimedia home platform (mhp) and related specifications. *Proceedings of the IEEE*, v. 94, n. 1, p. 237–247, Jan 2006. ISSN 0018-9219.

SARAIVA, D. et al. Architecting a model-driven aspect-oriented product line for a digital tv middleware: a refactoring experience. In: *Proceedings of the 4th European conference on Software architecture*. Berlin, Heidelberg: Springer-Verlag, 2010. (ECSA'10), p. 166–181. ISBN 3-642-15113-2, 978-3-642-15113-2.

SCHMIDT, D. C.; BUSCHMANN, F. Patterns, frameworks, and middleware: their synergistic relationships. In: *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003. p. 694–704. ISBN 0-7695-1877-X.

SOARES, L. F. G.; BARBOSA, S. D. J. *Programando em NCL 3.0: desenvolvimento de aplicações para o middleware Ginga*. Rio de Janeiro: Elsevier, 2009. ISBN 9788535234572.

SOARES, L. F. G.; RODRIGUES, R. F.; MORENO, M. F. Ginga-ncl: the declarative environment of the brazilian digital tv system. *JBCS: Journal of the Brazilian Computer Society*, SBC - Sociedade Brasileira de Computação, v. 13, n. 1, p. 37–46, March 2007. ISSN 0104-6500.

TAHVILDARI, L.; KONTOGIANNIS, K. A software transformation framework for quality-driven object-oriented re-engineering. In: *Proceedings of the International Conference on Software Maintenance (ICSM'02)*. Washington, DC, USA: IEEE Computer Society, 2002. p. 596–605. ISBN 0-7695-1819-2.

TOKUDA, L.; BATORY, D. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, Springer Netherlands, v. 8, p. 89–120, 2001. ISSN 0928-8910. 10.1023/A:1008715808855.

WU, Y. et al. Overview of digital television development worldwide. *Proceedings of the IEEE*, v. 94, n. 1, p. 8–21, Jan 2006. ISSN 0018-9219.