UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Tiago Stein D'Agostini

Adaptadores de Cenário como Técnica de Programação Orientada a Aspectos

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Antônio Augusto Medeiros Fröhlich
Orientador

Florianópolis, Julho de 2005

Adaptadores de Cenário como Técnica de Programação Orientada a Aspectos

Tiago Stein D'Agostini

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, área de concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____

Raul Sidnei Wazlawick

Banca Examinadora

_____

Antônio Augusto Medeiros Fröhlich

Orientador

_____

Olinto José Varela Furtado

_____

Raul Sidnei Wazlawick

_____

Leandro Buss Becker

Sumário

Abstract

Along the evolution of software engineering, several techniques were proposed in order to make Separation of Concerns easier and also to improve the efficiency on software development. This work targets two of these techniques: Aspect Oriented Programming and Static Metaprogramming. The focus is on the comparison of two techniques of Aspect Oriented Programming: Scenario Adapters and Aspect Oriented Programming by Code Weavers. Scenario Adapters were introduced as an alternative mechanism to apply aspects to software component at a time aspect weavers where yet in their early stage of development. The objective of this comparison is primarily to identify possible advantages on exchanging Scenario Adapters by weaver based Aspect Oriented Programming. Since Scenario Adapters were originally implemented as static metaprogrammed construct, this work also investigated the main issues related to the deployment of code weavers on that kind of code. The result of this investigation is a C++ `template` Preprocessor, capable of converting full featured `C++` code into simplified C++ code by executing eventual metaprogram during a first compilation stage. The comparative analysis of the techniques is performed on four Aspect Oriented Programming case studies. The evaluation was based on clarity of code, quantity of code developed and the likeness of producing flaw code with these techniques. The results of this analysis point on the direction that there is no absolute superiority of any technique above the other. In other words, there is no reasons for exchanging Scenario Adapters by weaver based Aspect Oriented Programming.

Resumo

Na evolução e no aprimoramento da engenharia de software, várias técnicas foram desenvolvidas com fins de facilitar a Separação de Conceitos, o reuso de código e a produtividade no desenvolvimento de programas. Este trabalho é focado em duas técnicas desenvolvidas com esses fins: Programação Orientada a Aspectos e Metaprogramação Estática. Mais especificamente, o trabalho é focado na comparação entre duas técnicas de Programação Orientada a Aspectos: Adaptadores de Cenário e Programação Orientada a Aspectos usando de *weavers* de código. A técnica de Adaptadores de Cenário, ambientada no escopo de sistemas operacionais, foi desenvolvida em uma época em que ainda não existiam *weavers* com características adequadas para uso neste escopo. O objetivo da comparação é principalmente identificar eventuais vantagens na substituição de Adaptadores de Cenário por Programação Orientada a Aspectos por meio de *weavers*. Para que esta análise pudesse ser realizada, também foi desenvolvido um estudo da viabilidade e solução sobre o uso de *weavers* de aspecto junto à Metaprogramação Estática e Classes Parametrizadas da linguagem C++. Deste estudo resultou a criação de um pré-processador de `templates` de C++. A comparação entre as técnicas foi desenvolvida em quatro estudos de caso de Programação Orientada a Aspectos. Os critérios de comparação basearam-se na clareza de código, quantidade de código desenvolvido e propensão a erro no desenvolvimento deste código. Os resultados desta comparação apontam que não existe superioridade marcante de uma das técnicas sobre a outra, não justificando a substituição de Adaptadores de Cenário por ferramentas de Programação Orientada a Aspectos baseada em *weavers* hoje disponíveis.

Capítulo 1

Introdução

Dentre várias facetas da engenharia de software, o controle sobre a complexidade dos programas de computadores é merecedor de destaque, tendo ajudado a moldar os métodos e ferramentas que dominam o desenvolvimento de software atualmente. Vários métodos e técnicas para administração da complexidade do software, bem como a complexidade do processo de desenvolvimento do software, foram propostos como partes integrantes de estratégias de engenharia de software tais como Projeto Baseado em Famílias *(Family-Based Design)* [26], Orientação a Objetos *(Object-Orientation)* [41], Programação Generativa *(Generative Programming)* [3], Desenvolvimento de Sistemas Orientados à Aplicação *(Application-Oriented System Design)* [5]. Dois fatores comuns a estas estratégias são a idéia de Separação de Conceitos *(Separation of Concerns)* [33] e a *modularização* do software, ambos com foco no controle da complexidade do software e de seu desenvolvimento.

Em sua grande maioria, as técnicas de engenharia de software, para serem efetivas no objetivo de manejo da complexidade do software [1], necessitam de ferramentas e linguagens que suportem seus conceitos. Tais ferramentas, mesmo quando não estritamente necessárias para a aplicação da técnica, são importantes para que o custo da aplicação da técnica não ultrapasse os benefícios obtidos pela mesma. De fato, a falta de suporte adequado por parte das linguagens de programação é uma das prováveis causas de que Projeto Baseado em Famílias [26] não tenha atingido nos anos 70, a aceitação que a aparentada Linhas de Produção de Software *(Software Product Lines)* [42] atinge nos dias de hoje.

Este trabalho engloba as técnicas de Programação Orientada a Aspectos *(Aspect*

[1] Complexidade de software é aqui tratada no sentido da complexidade de um software pontual, no mesmo tipo de complexidade tratada por Projeto por Famílias ou Orientação a Objetos

Oriented Programming) [23] e Metaprogramação Estática *(Static Metaprogramming)* [27]. O contexto deste trabalho é relacionado ao sistema operacional EPOS, desenvolvido por Fröhlich [5] como laboratório de Projetos de Sistemas Orientados a Aplicação *(Application Oriented System Design)*. O principal objetivo deste trabalho é analisar o uso da técnica de programação orientada a aspectos por meio de *Weavers* de código como substituto à tecnologia de Adaptadores de Cenário *(Scenario Adapters)* [8] nascida com o Projeto de Sistemas Orientados à Aplicação e usada no sistema *EPOS*. Analisando os resultados do desenvolvimento de aspectos em ambas as técnicas e de sua aplicação, o objetivo aqui é determinar os pontos fortes e fracos de ambas as técnicas quanto ao desenvolvimento de programas de aspectos respeitando as limitações do domínio de software básico, isto significa analisar as possibilidades de implementação sobre linguagens adequadas ao desenvolvimento de software básico. Como resultado final almeja-se a obtenção de um veredicto sobre a necessidade ou não do uso de linguagens de Programação Orientada a Aspectos e seus respectivos *weavers*, para tratar de questões ortogonais ao domínio, i.e "aspectos".

O domínio do desenvolvimento de software básico apresenta características que o diferenciam do desenvolvimento de software aplicativo. Dentre estas características estão as quais a baixa disponibilidade de poder de processamento e limitações no consumo de memória. A não possibilidade de abstrair questões tais como alocação de memória, tratamento de interrupções e acesso coerente ao hardware, tarefas que ficam sob responsabilidade do sistema operacional e dependem da arquitetura alvo, aumenta ainda mais as dificuldades do desenvolvimento de software neste domínio. Uma falha de execução em um sistema operacional tem repercussões sobre todas as camadas de software colocadas sobre o mesmo, este não podendo delegar a resolução de problemas a nenhuma camada de software mais abaixo. O escopo de sistemas dedicados embutidos aumenta ainda mais as restrições quanto a memória, tamanho de código e desempenho que já eram presentes do escopo de sistemas operacionais. Tais características podem influenciar técnicas e ferramentas utilizadas no desenvolvimento do software, incluindo a seleção das linguagens de programação. Para que uma linguagem de programação seja apta ao desenvolvimento de software básico é necessário que a mesma exponha ao programador a capacidade de controlar as características já citadas. Deste modo a linguagem deve permitir acesso direto ao hardware, incluindo memória, portas de I/O e registradores. Estas restrições inviabilizam o uso de linguagens populares no desenvolvimento de aplicativos como JAVA, que apesar de possuir extensões de suporte tanto à Metaprogramação quanto à Programação Orientada a Aspectos, não provê os mecanismos necessários para lidar de forma direta com o hardware. Deste modo as características inerentes ao desenvolvimento de software básico tem seu maior peso neste trabalho, no que diz respeito a

escolha das linguagens de programação e *weavers* de código adequados. As linguagens escolhidas para este trabalho são C++ [29], como linguagem de programação principal e AspectC++ [17] como linguagem e API de programação de aspectos.

No decorrer deste trabalho serão abordadas comparações teóricas entre Adaptadores de Cenário e Programação Orientada a Aspectos por meio de *weavers*. A complexidade de implementação de aspectos em ambas as técnicas serão analisadas sob a disponibilidade de ferramentas e técnicas adequadas ao escopo de sistemas operacionais. Serão também apresentados estudos de caso ilustrando características observadas em ambas as técnicas, resultando em uma avaliação das dificuldades do desenvolvimento de determinados aspectos em cada uma das técnicas.

Este trabalho trata tambêm, de demandas associadas ao ferramental necessário para realização deste estudo. Estas demandas referem-se a uma solução para a incapacidade do *weaver* de AspectC++ de aplicar aspectos em código *template*. A solução desta incapacidade é considerada relevante na determinação da validade do uso de "weavers"de código junto a C++. Para tanto é desenvolvido um ferramental visando a resolução de Metaprogramação Estática e classes parametrizadas de C++ em um estágio a parte do resto do processo de compilação. Este ferramental inclui modificações no compilador C++ e ferramentas de transformação de código.

Capítulo 2

A Separação de Conceitos e seu Legado: Programação Orientada a Aspectos

Dentre os conceitos propostos pela engenharia de software, um conceito que gerou várias ramificações e está presente como parte de outras técnicas é a idéia de Separação de Conceitos *(Separation of Concerns)* [33]. As implicações desta idéia são vistas na grande maioria das metodologias de desenvolvimento de software das últimas décadas. A necessidade de suporte a visões cada vez mais refinadas desta idéia tem levado as linguagens de programação em níveis cada vez mais complexos e capazes de lidar com problemas em níveis de abstração cada vez mais altos. Suporte por parte de linguagens de programação a metodologias e técnicas como Tipos Abstratos de Dados *(Abstract Data-Types)*, Orientação a Objetos *(Object-Orientation)* entre outras surgiram deste modo.

Uma das ramificações recentes deste conceito e que é de especial interesse neste trabalho, é a separação de conceitos não funcionais ou ortogonais ao domínio do problema. É sabido que a codificação de conceitos de forma localizada traz vantagens para o software e seu desenvolvedor. Por exemplo, um sistema cujos componentes devam apresentar persistência em um banco de dados pode obter este efeito através de um único componente de software responsável por tal capacidade. Deste modo se evita a necessidade de escrever código relativo a persistência em cada uma das classes do sistema. A concentração desta tarefa em um único componente diminui a quantidade de código a ser escrito ou mantido e facilita seu reuso. Estas são apenas algumas das vantagens que a separação de conceitos nos traz.

Não é raro um sistema ser projetado de forma elegante e com um bom nível de separação de conceitos, mas durante seu desenvolvimento, receber novas características ou

alterações em seu projeto. Tais alterações podem incluir operação sobre objetos de forma remota pela rede, concorrência ou verificação de erros e consistência de dados. Estas características tendem a ser implementadas de forma espalhada pelo código. O tratamento deste tipo de conceito por meio de técnicas que objetivam primariamente promover a separação de entidades funcionais do domínio pode apresentar dificuldades. No caso de Orientação a Objetos este tratamento é feito pela especialização de classes para o caso da ocorrência da característica não funcional em questão. Deste modo, uma classe de um *framework* poderia ser especializada para prover uma versão com capacidade de execução em múltiplos fluxos de execução concorrentes. A necessidade de especializar classes para o caso da ocorrência de cada uma das propriedades não funcionais de um sistema pode gerar um crescimento explosivo do número de artefatos de software do sistema ou *framework*. Encapsular estes conceitos em um ponto único pode ser problemático. Isto porque estes conceitos atravessam os limites das abstrações de domínio usualmente identificáveis e demarcáveis por meio de técnicas de encapsulamento tradicionais.

A Programação Orientada a Aspectos [23] apresenta uma proposta de solução para modularização deste tipo de conceito. Os conceitos ortogonais ao domínio do problema são identificados como aspectos do sistema. A proposição da Programação Orientada a Aspectos de encapsular aspectos como artefatos de software independentes e efetuar combinação dos mesmos com abstrações alvo por meio de *weavers* provê respostas de como implementar a separação de conceitos, mas muito pouco sobre como identificar estes conceitos de uma forma eficiente durante o desenvolvimento. O processo de análise e engenharia de domínio é comumente de grande importância em metodologias como Desenvolvimento Orientado a Objetos e ainda é um ponto em evolução na Programação Orientada a Aspectos. Muitas vezes o conceito de Programação Orientada a Aspectos é associado ao processo de *weaving* de programas de aspectos, representado na figura 2.1. Neste processo, uma ferramenta de composição de código denominada de *weaver* é usada para combinar o código de um aspecto ao código de uma abstração específica.

Várias técnicas foram desenvolvidas na tentativa de prover um mecanismo de identificação e separação de propriedades não funcionais de um sistema. Algumas destas técnicas nem sempre são referidas como Programação Orientada a Aspectos, mas abordam os mesmos problemas e podem ser consideradas conceitualmente semelhantes, ou aparentadas à Programação Orientada a Aspectos. Dentre estas técnicas pode ser citada a Programação Orientada a Sujeitos *(Subject Oriented Programming)* [21] que apresenta uma visão de desenvolvimento que permite identificar as características não funcionais do sistema, bem como apresenta uma técnica de modularização e implementação destes aspectos. Esta técnica a princípio pode parecer diferente da

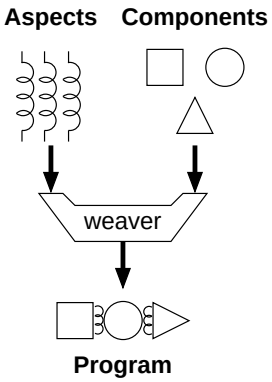

Figura 2.1: Aplicação de Aspectos por Weavers (Conforme Fröhlich[5]).

apresentada por Programação Orientada a Aspectos, mas fundamentalmente tem um resultado final semelhante. A figura 2.2 demonstra a visão geral da composição de código em Programação Orientada a Sujeitos. Nesta um compositor de código é usado para combinar duas visões de uma determinada abstração, incluso a resolução de possíveis conflitos de nomenclatura e interface de ambas as visões. Ao comparar esta figura com a correspondente para Programação Orientada a Aspectos por weavers 2.1 é visível a semelhança dos dois modelos. Através da modularização das abstrações do sistema em visões de cada cliente da abstração, Programação Orientada a Sujeitos possibilita a modularização de características como concorrência, persistência, entre outras, em visões que não interferem diretamente sobre outras. Esta técnica, entretanto, permite modularizações de características dos aspectos ortogonais ao domínio do problema, bem como apresenta uma solução no sentido de facilitar o desenvolvimento de programas por grupos de desenvolvimento separados entre si.

Outras técnicas de engenharia de software apresentam abordagens que possibilitam em maior ou menor grau a separação de conceitos funcionais e não funcionais de um sistema. Metodologias e ferramentas que suportem a construção e extensão de linguagens possibilitam o encapsulamento das características não funcionais de um sistema. Exemplo de técnicas que podem tratar de ao menos alguns aspectos são Filtros de Composição *(Composition Filters)* [12] e métodos baseados em metaprogramação. Filtros de composição surgem para tratar problemas de coordenação entre mensagens. Para tanto, introduzem o conceito de filtros de mensagens, pelos quais todas as mensagens de um sistema passam. Independentemente do método de implementação destes filtros, esta idéia pode ser usada para aplicar aspectos durante a passagem das mensagens pelos filtros. A linguagem OPENC++[10] representa uma abordagem baseada na metaprogramação. Esta linguagem possui mecanismos de metaprogramação que permitem

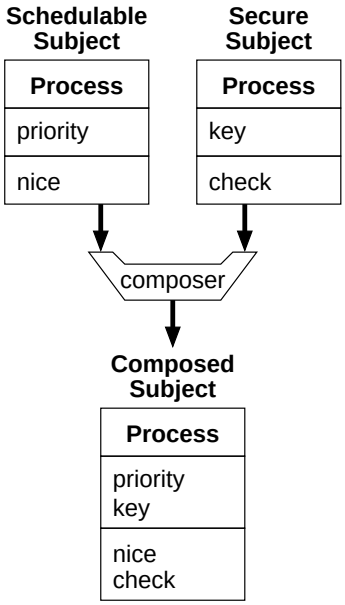

Figura 2.2: Programação por Sujeitos (Conforme Fröhlich[5]).

a definição de linguagens de domínio específico. As extensões definidas através de OPENC++ podem ser usadas para capturar características funcionais de um domínio, bem como algumas características não funcionais. Apesar de não prover os mecanismos de extensão de linguagem explicitamente como uma forma de captura de aspectos, pode ser usada com este fim. Ao capturar características não funcionais de um sistema por meio de uma extensão à linguagem, consegue-se o encapsulamento de um aspecto.

2.1 Mecanismos de Composição de Aspectos

O foco deste trabalho não está na identificação de aspectos durante o processo de análise e fatoração do domínio, mas sim nos mecanismos de descrição e composição de aspectos, usados para combinar os aspectos às abstrações funcionais do sistema. Abstrações que podem ser identificadas e modeladas com praticamente qualquer outra técnica de separação de conceitos. As linguagens de programação modernas têm predominantemente influência das técnicas de separação de conceitos funcionais e essa influência é refletida no amplo suporte a conceitos como funções, parametrização de código, herança e outros. O suporte direto desses mecanismos nas linguagens de programação e ferramentas afins, permitiu que técnicas de engenharia de software como Orientação a Objetos atingissem um significativo grau de sucesso na produção de software. Para que Programação Orientada a Aspectos seja bem sucedida em sua tentativa de tratar aspectos

não funcionais de um domínio, fazem-se necessários mecanismos eficientes para aplicá-la. Algumas das técnicas já citadas e que possuem uma semelhança com o conceito de Programação Orientada a Aspectos, possuem seus próprios mecanismos de composição e aplicação de código, como as regras de composição usadas em Programação Orientada a Sujeito, filtros de mensagens usados em *Composition Filters* e estratégias de percorrimento usando *visitors* como em Programação Adaptativa *(Adaptive Programming)* [13].

Qualquer que seja a técnica usada para suportar a idéia de aspectos, é ideal que a mesma alcance objetivos de mínimo acoplamento entre os componentes funcionais e os aspectos; sendo desejável também uma interação o mínimo invasiva possível na adição dos aspectos ao código existente das abstrações. Uma interação ou operação invasiva sobre um artefato de software é aquela em que os detalhes do artefato são expostos, manuseada ou precisa ser conhecida além de sua interface. Ambas condições são derivações lógicas da idéia de separação de conceitos. Outro fator importante é a robustez do processo de composição, visto que a aplicação de um aspecto que resulte em efeitos colaterais nos componentes alvos não é satisfatória. Este último fator vem parcialmente de encontro com a necessidade de separação de conceitos, visto que um aspecto e uma abstração confeccionados de forma completamente independente tendem a apresentar maiores chances de incompatibilidade do que um aspecto desenvolvido com uma abstração ou um conjunto de abstrações específicas em mente. Este problema também pode ocorrer na interação entre dois aspectos aplicados sobre o mesmo sistema, mas com efeitos conflitantes. Um nível de compromisso e equilíbrio é necessário para lidar com este conflito, visto que a combinação de qualquer aspecto em qualquer abstração apresenta chances de efeitos colaterais indesejáveis.

2.2 Pontos de Junção

Os mecanismos de composição de aspectos levam à necessidade da elaboração do conceito de Ponto de Junção *(Join Point)* [23]. O termo “Ponto de Junção”é, como o nom eindica, um ponto onde duas partes de código se juntam. As duas partes em questão são o código funcional e as características não funcionais e ortogonais ao domínio, ou seja, os aspectos. Um exemplo desta idéia é quando um código tradicional é separado em duas partes; uma contendo todas as estruturas funcionais do problema e outra com os conceitos ortogonais a este problema, mas necessários para o correto funcionamento do programa. Em uma destas partes, possivelmente em ambas, existirão referências para código pertencente à outra parte. Estas referências entre as partes separadas do código são os pontos de junção. Em outras palavras, os pontos de junção

podem ser vistos como os pontos de estabelecimento de interface entre o código de um programa de aspecto com o código de sua abstração alvo. Diferentes mecanismos de composição podem suportar diferentes tipos de pontos de junção, como a estrutura hierárquica de classes, chamadas de função ou conjunto de membros de uma classe.

Um tipo específico de ponto de junção que é importante neste trabalho e está presente em código Orientado a Objetos é chamado de Pontos de Junção em Mensagens *(Message Join Points)* [3] ou também de Ponto de Junção de Operação *(Operation Join Point)* [4]. Basicamente esta categoria de pontos de junção cobre a troca de mensagens entre objetos e pode ser implementada acoplando-se código na chamada ou execução do método, bem como no retorno do mesmo. É digno de nota que este tipo de ponto de junção é importante em Programação Orientada a Objetos e que pode cobrir grande parte dos casos de decomposição de aspectos em sistemas planejados segundo Orientação a Objetos. Exemplos de aspectos que comumente podem ser tratados apenas com o uso deste tipo de ponto de junção incluem concorrência, tratamento de erros, sincronização, entre várias outras. O embasamento no conceito de mensagens entre objetos presente em Orientação a Objetos é o principal responsável por esta situação. Um sistema desenvolvido deste modo tende a ser um sistema onde a maioria dos pontos relevantes, e conseqüentemente a maioria dos pontos de junção, são tratados por meio do mecanismo de mensagens entre objetos. A listagem 2.3 mostra um exemplo simplificado de como mensagens podem ser usadas como pontos de junção para aspectos. Esta implementação faz uso do mecanismo de polimorfismo de sub-tipo através de métodos `virtual`. A classe `Child` re-implementa o método `performAction()` de sua classe mãe `Super`, executando os métodos `on_entrance()` e `on_leave()` na entrada e saída da função. Ações de um programa de aspecto podem ser realizadas por meio destes métodos. A mensagem original, por sua vez, continua sendo processada entre as chamadas de `on_entrance()` e `on_leave()`. Através deste mecanismo as mensagens a objetos da classe `Super` são interceptadas e um programa de aspecto pode ser acionado.

Outro ponto de junção relevante sobre código Orientado a Objeto são as próprias classes. Este tipo de ponto de junção é usado junto a programas de aspecto que adicionem novos membros à classe alvo. Assim por exemplo é possível adicionar um membro a uma classe, com a função de prover a identificação unívoca de um determinado objeto dentre todos os objetos de um sistema. Uma técnica capaz de trabalhar com mensagens como ponto de junção e com capacidade de agregar membros a classes é uma técnica suficientemente abrangente de Programação Orientada a Aspetos para ser usada em diversos casos de sistemas orientados a objetos.

```
class Super{

public:

 virtual  void performAction();

};


class  Child : public Super{

public:

  void on_entrance();
  void on_leave();

  void performAction()
  {
        on_entrance();
        Super::performAction();
        on_leave();
  }
};
```

Figura 2.3: Interceptação de mensagens como ponto de junção.

2.3 Composição por *Weavers*

Atualmente os mecanismos de composição de maior destaque na Programação Orientada a Aspectos são os *code weavers* [23]. Estes têm como função "costurar"um código junto aos pontos de junção em outro código, sem necessitar de implementações especiais no código alvo em questão. Em outras palavras, os *weavers* misturam o código do aspecto com o código das abstrações alvo, usando para isso expressões de ponto de corte *(Pointcut)* como referência. Um ponto de corte é basicamente um conjunto de pontos de junção determinados através de uma expressão. Normalmente *weavers* de apectos fazem uso de uma linguagem de descrição de aspectos e de uma API ou dialeto de descrição de pontos de junção baseado em uma linguagem de programação. Exemplos de linguagens ou extensões de linguagens para Programação Orientada a Aspectos que opearm deste modo são ASPECTJ[15], ASPECTC[14] e ASPECTC++[17]. A linguagem de descrição de aspectos e a ferramenta de composição podem ser desenvolvidas com a descrição de diversos tipos de pontos de junção em mente. Como resultado, estas técnicas podem suportar uma variedade de tipos de pontos de junção diferentes, basicamente sendo possível suportar qualquer ponto de junção descritível na gramática da linguagem.

O desenvolvimento de *weavers* e de linguagens de definição de ponto de corte, tal como ASPECTJ e ASPECTC++, são ambas atividades complexas e de alto custo, principalmente quando desenvolvidas com fins de aplicar aspectos a linguagens complexas tal como C++. Prova disso é o período extenso de desenvolvimento destas ferramentas e o fato de que ainda hoje algumas destas não suportam todas as características da linguagem alvo. Exemplo disto são as limitações que ASPECTC++ apresenta na interação com código *template*, bem como o suporte limitado a algumas características de C++ como sobrecarga de operadores, mesmo depois de anos de desenvolvimento.

É importante ressaltar que o mecanismo de *weaving*, apesar de ser comumente visto como sinônimo de Programação Orientada a Aspectos, não é a única técnica de composição de aspectos. Várias linguagens de programação modernas possuem mecanismos que permitem aplicação de aspectos em determinadas condições de uso e quando suportados por artefatos de software construídos com este fim. Um exemplo de situação onde aspectos podem ser aplicados com mecanismos comuns de linguagem é a interceptação de mensagens como ilustrado anteriormente. Com o uso de alguns Padrões de Projeto *(Design Patterns)* [18], como Decoração *(Decorators)* e Estratégias *(Strategies)* por exemplo, é possível aplicar alguns tipos de aspectos a um sistema. A figura 2.4 ilustra o padrão de projeto Decoração, que pode ser usado na aplicação de aspectos que envolvam a inserção de novos membros em classes. Exemplos de técnicas que têm sucesso na aplicação de programas de aspecto por meio do suporte de artefatos de software são *Adaptadores de Cenário* [8] e a técnica descrita por Musser em *A Metaprogramming Approach to Aspect Oriented Programming in C++* [25]. Nesta última técnica, os aspectos são implementados como metaprogramas e são combinados e ativados também por meio de metaprogramação estática. Entretanto a abordagem dos *weavers* é especialmente atraente para linguagens sem capacidades semelhantes à Metaprogramação Estática de C++ ou a flexibilidade de interação com o ambiente de execução apresentada por linguagens como SMALTALK [9] e PYTHON[11].

A mesma falsa associação de Programação Orientada a Aspectos com *weavers*, citada anteriormente, criou uma situação merecedora de atenção no desenvolvimento de software: ferramentas de Programação Orientada a Aspectos são usadas como mecanismos de reparo e manutenção de códigos, usualmente sem qualquer associação com os conceitos de Programação Orientada a Aspectos. Esta situação parece indicar que a Programação Orientada a Aspectos é freqüentemente associada com suas ferramentas e não com seus conceitos. O porquê disto não é facilmente explicável, mas é da opinião do autor que tenha origem na escassez de metodologias de identificação e projeto de aspectos firmemente associados ao conceito de Programação Orientada

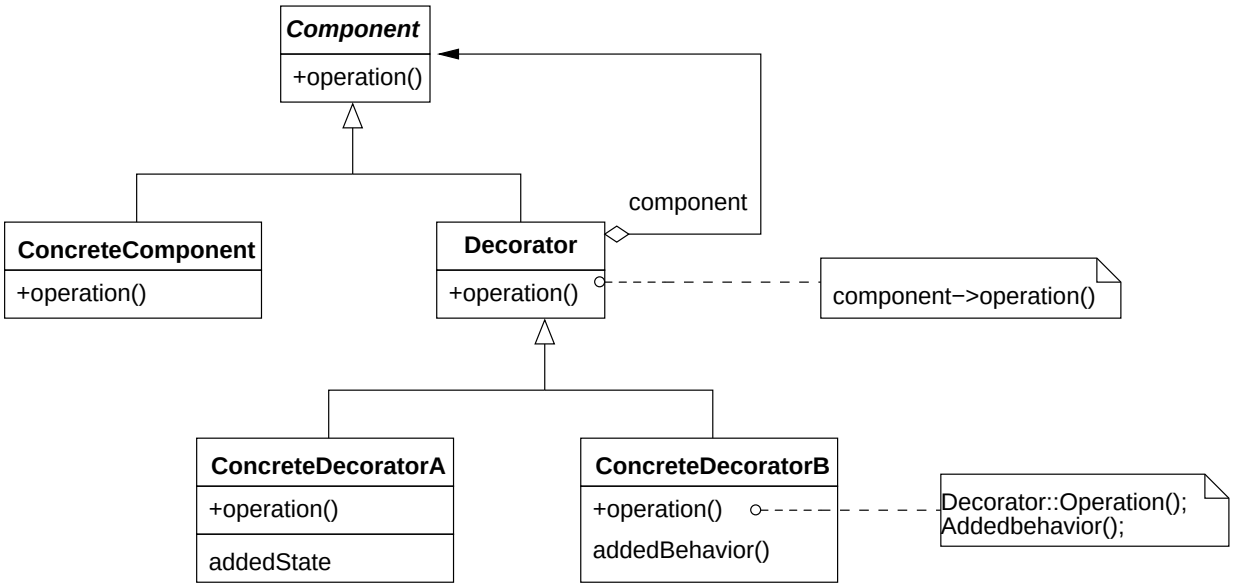

Figura 2.4: Estrutura do Padrão de Projeto Decoração (Conforme Gamma[18]).

a Aspectos desde seu princípio.

Várias técnicas e ferramentas de Programação Orientada a Aspectos estão presentes na indústria e no meio acadêmico hoje em dia. Muitas destas técnicas não pretendem ser consideradas como tal, mas podem ser vistas sob um prisma que assim as favoreça. Neste trabalho duas técnicas de programação orientadas a aspectos estão sob foco: a composição por *weaving* representada pela linguagem ASPECTC++ e a Adaptadores de Cenário[5]. O escopo deste trabalho é concentrado em sistemas operacionais, mais especificamente sistemas dedicados de pequeno porte, ou embutidos. Ambos, Adaptadores de Cenário e ASPECTC++, foram desenvolvidos para aplicar aspectos em sistemas operacionais. Este escopo apresenta um conjunto de dificuldades específicas que necessitam de especial atenção. Ambas as técncias já demonstraram em trabalhos nas quais foram utilizadas, que são capazes de adequar-se a estes requisitos.

2.4 Adaptadores de Cenário

A técnica de Adaptadores de Cenário foi concebida junto a Projeto de Sistemas Orientados à Aplicação[5], como um mecanismo de Programação Orientada a Aspectos sem o uso de *weavers* de código. A visão geral de Adaptadores de Cenário é representada na figura 2.5. Adaptadores de Cenário são artefatos de software que permitem a interceptação de mensagens direcionadas a uma abstração e a ativação de um conjunto de aspectos que definem um cenário. O

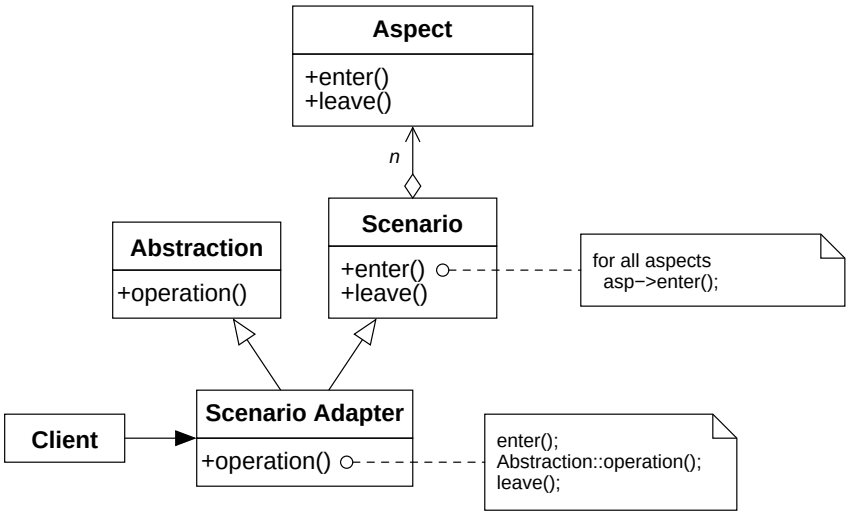

Figura 2.5: Visão geral de Adaptadores de Cenário (Conforme Fröhlich[5]).

acesso a uma abstração é feito através de um adaptador que combina a abstração com um conjunto de aspectos. A ativação dos aspectos pode ser feita durante métodos de entrada ou saída, ativados ao redor da chamada de método da abstração.

Adaptadores de Cenário são capazes de prover a composição de aspectos com mensagens como pontos de junção (*message join-points*) que, como já fora discutido na seção anterior, representam apenas uma parte dos pontos de junção possíveis, mas que são de grande importância no paradigma de Orientação a Objetos. Pode-se implementar Adaptadores de Cenário de modo que seja possível a introdução de novos membros nas classes alvo dos aspectos, aumentando assim a diversidade de aspectos que podem ser implementados com Adaptadores de Cenário. Uma implementação com esta capacidade será discutida mais adiante neste trabalho.

Quando Projeto de Sistemas Orientados à Aplicação foi desenvolvida, não se dispunha de *weavers* de aspectos para linguagens adequadas ao desenvolvimento orientado a objetos de sistemas operacionais. ASPECTC++, que é uma implementação de *weaver* de aspectos adequada ao uso no desenvolvimento de sistemas operacionais, ainda se encontrava em estágio inicial de desenvolvimento. Há época, esperava-se que logo fosse desenvolvida uma linguagem de Programação Orientada a Aspectos e *weaver* maduros e capazes de operarem sobre uma linguagem adequada ao desenvolvimento de sistemas operacionais. Ao mesmo tempo era esperado que este *weaver* introduzisse um método mais flexível e de maior clareza no desenvolvimento de aspectos. Isto alcançado, Adaptadores de Cenário seriam obsoletos e substituíveis por uma linguagem de Programação Orientada a Aspectos, tal como é a prática predominante na Programação Orientada a Aspectos no desenvolvimento de software aplicativo. A validade desta hipótese será discutida

nos estudos de caso realizados neste trabalho e descritos em seção posterior.

Capítulo 3

Metaprogramação Estática e Adaptadores de Cenário

Neste capítulo serão tratadas uma implementação específica de Adaptadores de Cenário e a tecnologia sobre a qual esta implementação é baseada: Metaprogramação Estática. O termo Metaprograma é definido por Veldhuizen[40] como:

"Um metaprograma é um programa que manipula outros programas."

Decorre desta definição, que Metaprogramação é programação que serve para descrever ou manipular um programa. Exemplo de ferramenta de metaprogramação são os compiladores, manipulando código em uma linguagem com fins de convertê-lo em código em outra linguagem. Os mecanismos de macros, presentes em linguagens como C e C++ são também exemplos de metaprogramação, pois estas produzem código, alterando a semântica dos programas onde são usadas. A Programação Orientada a Aspectos, na sua forma de uso com ferramentas de *weaving* é por si só uma forma de metaprogramação, visto que os weavers operam sobre código fonte de programas, alterando-os. Um tipo de metaprogramação, que apesar de não ser o foco deste trabalho é bastante conhecida e merecedora de citação é a Reflexão. É chamada de reflexão a metaprogramação em que o programa alvo e executor são o mesmo. Reflexão é definida por Smith [6] como:

"Reflexão: é a capacidade de uma entidade em representar, operar em, e de outros modos lidar consigo mesma do mesmo modo que ela representa, opera em, ou lida com sua principal preocupação."

Um exemplo de linguagem com capacidades de reflexão é SMALTALK. Em SMALTALK as classes são representadas por objetos de uma metaclasse, que por sua vez também

são objetos manipuláveis. Modificando as metaclasses é possível modificar o modelo de objetos da linguagem. Algumas outras linguagens que apresentam suporte à Reflexão Computacional são OBERON-2[38], IO[36] e extensões da linguagem JAVA[34]. Com o uso de Reflexão, um sistema pode ser dividido em dois níveis: um "metanível"e um nível base. Buschmann [35] descreve estes níveis como:

" Um metanível provê informações sobre propriedades de um sistema e faz o software ter noção de si mesmo. Um nível base inclui a lógica da aplicação."

A definição da interface de operação deste "metanível"leva à criação de Protocolos de Metaobjetos *(Metaobject Protocols)* [39]. Usando deste protocolo e através de mecanismos tais como a derivação de novas metaclasses a partir das já existentes, a semântica das metaclasses de um sistema pode ser modificada e adaptada conforme necessário.

O momento em que um metaprograma é executado constitui um ponto diferenciador entre as diversas técnicas de metaprogramação. A metaprogramação em tempo de execução traz ao programador uma sensação de flexibilidade ampliada e de adaptatividade sem grandes restrições. O modelo de metaproprogramação em tempo de compilação tende a passar uma aparência de menor flexibilidade quando comparado ao modelo dinâmico, visto existirem casos em que a adaptação de código passa a ser necessária a partir da obtenção do valor de uma variável em tempo de execução. Esta situação pode surgir, por exemplo, na carga de módulos externos do programa. Todavia, mecanismos de linguagens ativados em tempo de execução algumas vezes apresentam desvantagens quando comparados aos seus correspondentes resolvidos em tempo de compilação. A flexibilidade do modelo dinâmico traz consigo o custo de mecanismos de gerência e da aplicação da metaprogramação em tempo de execução. Estes mecanismos podem implicar perda de performance e aumento do tamanho do programa resultante. Estas perdas podem ser negligenciáveis em muitos casos, especialmente quando a flexibilidade em tempo de execução é estritamente necessária. Todavia, a aplicação desses mecanismos pode ser mero desperdício de recursos quando esta flexibilidade não tem um uso.

O ambiente dos sistemas embutidos dedicados é um caso onde técnicas dinâmicas são normalmente desfavorecidas. Os recursos limitados de muitos sistemas embutidos tornam crítica a perda de performance e o aumento no consumo de recursos. Um exemplo de técnica usada no desenvolvimento de sistemas de grande porte sem a preocupação de impactos negativos é o Polimorfismo por Subtipo implementado por ligação dinâmica de métodos. Esta técnica auxilia no desenvolvimento de software, mas tem custos que não podem ser evitados completamente. Uma chamada de um método `virtual` em linguagens como C++ é ligeiramente mais lenta que

uma chamada comum de função, devido a busca na tabela de métodos `virtuais` pela função que deve ser executada. Uma chamada comum de função, por sua vez, é mais lenta que o uso de um método `inline`, no qual o código da função é inserido no local da chamada do mesmo. Além da perda de performance devido ao processamento extra, existe um aumento no tamanho do código objeto, resultante dos mecanismos de ligação tardia e da tabela de métodos virtuais. Nos casos onde o dinamismo não for estritamente necessário, é possível aplicar o Polimorfismo Paramétrico [7], através de mecanismos tais como classes parametrizadas providas por `templates` de C++. Estas técnicas são úteis, por exemplo, quando a flexibilidade é necessária apenas com fins de permitir adaptabilidade e "configurabilidade"entre componentes durante o desenvolvimento do sistema. O mesmo tipo de problema pode ser encontrado no uso de técnicas de metaprogramação em tempo de execução, quando no escopo de sistemas operacionais e embutidos.

O exemplo de metaprogramação mais comumente associado com o conceito de Metaprogramação Estática é representado pela linguagem C++ [29]. A linguagem C++ tem capacidades de metaprogramação [27, 40] através da abordagem de programação em múltiplos níveis. Um nível representa o código dinâmico que é compilado e executado após a carga do programa. O outro nível do programa é executado em tempo de compilação, tendo efeitos sobre o código do nível dinâmico. Diversos mecanismos da linguagem C++ são responsáveis por sua capacidade de metaprogramação, dentre eles a avaliação parcial de expressões em tempo de compilação, a designação de *pseudônimos* para tipos e principalmente o mecanismo de *templates*. As vantagens de desempenho propiciadas por um sistema de metaprogramação estática mostram-se ainda mais úteis nos escopos onde a linguagem C++ é comumente utilizada, nos quais não raramente são requeridos programas pequenos e de boa performance. Deste modo, Metaprogramação Estática pode ser usada como mecanismo para auxiliar no desenvolvimento de sistemas de alta performance, como é demonstrado por Veldhuizen [31] com Bibliotecas Ativas *(Active Libraries)*.

Exemplos de Metaprogramação Estática bastante conhecidos são o metaprograma fatorial da figura 3.1 e o metaprograma de condicional estática na figura 3.2. No primeiro programa, um *template* parametrizado por um número inteiro tem sua definição baseada na recursão para o mesmo *template*, o qual é parametrizado com o valor original reduzido em um (1). Um caso especial de parada é fornecido pela especialização do `template` para o argumento zero (0). As regras de resolução de *templates* na linguagem C++ fazem com que, na presença de uma especialização que combine com o argumento em questão, esta seja utilizada no lugar do *template* genérico. Analisando a sintaxe deste código percebe-se que o método de programação estática em C++ tem similaridades com o paradigma de programação funcional, uma característica con-

```
template<int n>
class Factorial {
  public:
  enum{ RET = Factorial<n − 1>::RET ∗ n };
};

template<>
class Factorial<0> {
  public:
  enum { RET = 1 };
};

int main()
{
   return Factorial<4>::RET;
}
```

Figura 3.1: Programa Fatorial Metaprogramado.

trastante ao paradigma imperativo usado normalmente no código dinâmico da linguagem. Esta dicotomia nos métodos de programação é, possivelmente, uma das causas responsáveis pela tradicional noção da sintaxe de metaprogramação em C++ como complexa e de pouca legibilidade, mas ajuda a traçar limites tangíveis entre os dois níveis de linguagem.

3.1 O Uso de Adaptadores de Cenário no Framework Metaprogramado de Sistema EPOS

Uma das possíveis implementações de Adaptadores de Cenário é a apresentada no sistema EPOS. Este sistema é uma implementação laboratório de Projeto de Sistemas Orientados a Aplicação. Nesta implementação, Adaptadores de Cenário são artefatos de software metaprogramados e são usados em conjunto com um *framework* de sistema do EPOS. A figura 3.3 ilustra esta implementação de Adaptadores de Cenário.

Na implementação em consideração, o cliente de uma abstração do sistema só pode acessá-la através do sistema de Adaptadores de Cenário. O acesso a uma abstração é feito de forma indireta através de um `Handle` parametrizado com a abstração desejada. O `Handle` é uma classe parametrizada, que exporta a interface combinada dos membros do sistema, de forma que o cliente pode invocar métodos de um objeto do tipo `Handle` do mesmo modo que ele faria com a abstração propriamente dita. O *framework* de sistema provê um mecanismo de exportação de

```
template<bool condition, typename Then, typename Else>
class IF
{
        public:
        typedef Then RET;
};


// specialization   for  condition=false

template<typename Then,typename Else>
class IF<false,Then, Else>
{
 public:
        typedef Else RET;
};


// ...

IF<(sizeof(int)>=4),unsigned int, unsigned long long>::RET integer;
```

Figura 3.2: Condicional metaprogramada.

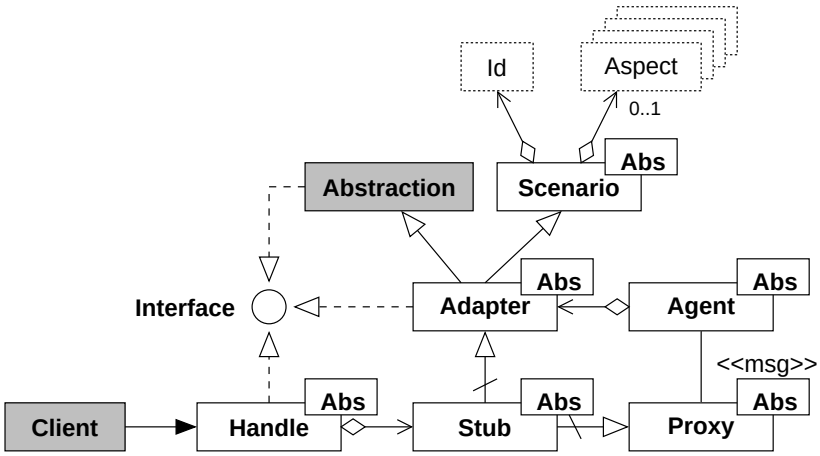

Figura 3.3: Adaptadores de Cenário como implementados no sistema EPOS (Conforme Fröhlich[5]).

```
template<class Imp>
class Adapter: public Scenario<Imp>, public Imp
{

  public:

  void performAction()
  {
        Scenario<Imp>::enter();
        Imp::performAction();
        Scenario<Imp>::leave();
  }

};
```

Figura 3.4: Implementação de um `Adapter`.

nomes entre `namespaces`, o qual atribui ao `Handle` um pseudônimo concernente à abstração em questão. Deste modo o usuário de uma abstração do sistema não sabe que está lidando com o encapsulamento da abstração, ao invés de com a abstração real.

O objeto do tipo `Handle` propaga as mensagens dirigidas à abstração até o `Adapter`, o qual combina um `Scenario` com a abstração propriamente dita. A combinação destes artefatos é feita através do mecanismo de herança múltipla de C++. O `Adapter`, que é parametrizado com a abstração alvo, intercepta as mensagens e chama métodos de `enter()` e `leave()` ao redor da execução do método original da abstração. Estes métodos correspondem às ações a serem tomadas na entrada e na saída do cenário de operação. A figura 3.4 ilustra um adaptador provido da mesma interface utilizada no exemplo de interceptação de mensagens. Existe uma diferença no processo de interceptação descrito aqui e o processo discutido anteriormente. Enquanto a implementação descrita na figura 2.3 faz uso de um polimorfismo dinâmico de subtipo, a implementação de `Adapter` tem a interceptação resolvida em tempo de compilação. A resolução da interceptação de mensagens em tempo de compilação elimina as penalidades sobre desempenho e tamanho de código associadas com métodos virtuais.

A classe `Scenario`, ilustrada na figura 3.5, representa a combinação de diversos aspectos ativos em um cenário de operação do sistema. Esta classe combina as implementações de todos os aspectos ativos para a abstração em questão através do mecanismo de herança múltipla. A definição de quais classes devem compor esta herança é feita através de um metaprograma representado na mesma figura. Este metaprograma recebe a abstração através do parâmetro da classe `Scenario` denominado `Imp`. O metaprograma terá como resultado as classes de implementação

dos aspectos quando estes estiverem ativos, ou uma classe vazia quando inativos. As informações necessárias para o processamento destes metaprogramas é suprida por Classes de Configuração Estática *(Traits)* do *framework* de sistema. É também na classe `Scenario` que os métodos de entrada e de saída do cenário são definidos, executando quaisquer ações relacionadas diretamente aos aspectos ativos.

O mecanismo descrito acima é estendido para dar suporte ao aspecto Invocação Remota de Métodos através de uma estrutura semelhante ao Padrão de Projeto Ponte *(Bridge)*, representada pelas construções de `Proxy` e `Agent`. O elemento `Proxy` repassa a invocação de métodos para um `Agent` através de um mecanismo de Invocação Remota de Objetos. O elemento `Agent`, por sua vez, repassa a mensagem ao `Adapter` de modo similar ao que é feito por `Handle` quando Invocação Remota de Objetos não é usada. Uma construção denominada `Stub` é usada para determinar se `Handle` repassará chamadas de métodos para `Adapter` ou para `Proxy`. A determinação de qual destas opções será usada é feita com base em informações de configuração do *framework*.

Pode-se observar que Adaptadores de Cenário fazem uso dos padrões de projeto Adaptador *(Adapter)* e Ponte e pode-se ver semelhanças com o padrão Decoração *(Decorator)*. Nota-se que Adaptadores de Cenário representam uma técnica de aplicação de aspectos que, embora eficaz no que se propõe, não apresenta as capacidades secundárias de correção e modificação de código que os mecanismos de *weaving* apresentam. Do mesmo modo os limites na aplicação de aspectos estão restritos a pontos de junção de mensagens. Para tanto, as interfaces das abstrações do sistema devem estar refletidas nas construções do *framework* tal como `Handle`.

A estruturação do sistema em um *framework* dá a capacidade de controlar características e relacionamento entre entidades, tais como a ordem de inicialização das mesmas, garantindo um funcionamento adequado do sistema. As informações providas pelo *framework* também podem ser usadas por suas abstrações ou pelos programas de aspectos. Dentre as diversas informações disponibilizadas pelo *framework* estão a identificação de tipos por meio de valores numéricos e um sistema de classes de configuração estática *(Traits)* que pode abrigar informações quanto a características configuráveis do *framework* bem como informações sobre que abstrações e aspectos estão selecionados para a geração do sistema. O *framework* pode restringir correlações de entidades e aspectos aos casos onde isto é adequado. Deste modo, por exemplo, o *framework* não permite que o aspecto Atomicidade, usado na implementação de suporte a concorrência, seja aplicado à classe `Mutex`, visto que esta aplicação não faria sentido. Do mesmo modo o *framework* garante que duas abstrações ou programas de aspecto mutuamente exclusivos não sejam aplicados

```
template<class Imp>
struct  Traits {
        static  const bool aspect=false;
};

template<>
struct  Traits <Abstraction>{
        static  const bool aspect=true;
};

template<bool active>
class Aspect{
        // Implements an aspect
};

template<>
class Aspect<false>{
        // Implements nothing
};


template<class Imp>
class Scenario: public Aspect<Traits<Imp>::aspect>,
{

public:
        void enter ();

        void leave ();

};
```

Figura 3.5: Implementação de um `Scenario`.

ou usados simultaneamente.

Uma característica da técnica de Adaptadores de Cenário que permitiu seu sucesso no EPOS é a capacidade de interceptar mensagens e aplicar os aspectos sem implicar perdas de desempenho outras que não as provenientes diretamente do programa de aspecto em si. Essa capacidade deve-se fundamentalmente ao uso de técnicas de Metaprogramação Estática, que permitem que relações entre artefatos de software sejam implementadas de um modo que o compilador possa eliminar quaisquer adições desnecessárias de código devidas ao processo de combinação dos artefatos. Para que isto seja possível, os mecanismos de avaliação de expressões em tempo de compilação e *inlining* de métodos são utilizados. O resultado final combina a abstração original com o programa de aspecto de uma forma livre de código resultante da combinação dos artefatos propriamente dita.

A técnica de Adaptadores de Cenário não é a única que implementa a aplicação de programas de aspecto através de Metaprogramação Estática. Outros trabalhos apresentam meios de aplicar aspectos usando os mecanismos de Metaprogramação Estática da linguagem C++ como ferramenta de composição. Um exemplo destes é o trabalho baseado na biblioteca metaprogramada BOOST apresentado por Musser [25]. Esta técnica permite aplicar aspectos usando, na maior parte, apenas classes e *templates* já providos pela biblioteca BOOST de extensão à STANDARD TEMPLATE LIBRARY (STL). A abordagem de Musser usa uma aplicação explícita do aspecto por parte do desenvolvedor da abstração. Este método é menos transparente e não promove uma separação entre aspectos e abstrações nos mesmos níves de Adaptadores de Cenário. O menor nível de transparência na aplicação dos aspectos deve-se à presença de código de ativação do aspecto na abstração, que é alvo do programa de aspectos. Este método também tem as mesmas limitações de Adaptadores de Cenário quanto aos pontos de corte suportados, não apresentando assim vantagens significativas sobre Adaptadores de Cenário.

Como já foi citado, OPENC++ pode fazer uso de metaprogramação por meio de Reflexão para alcançar objetivos de Programação Orientada a Aspectos. Apesar de tanto Adaptadores de Cenário quanto OPENC++ fazerem uso de metaprogramação, existem diferenças significativas no uso de Metaprogramação Estática entre elas. Enquanto OPENC++ faz uso de um sistema de Reflexão, incluindo MOPs, Adaptadores de Cenário usam de Metaprogramação Estática apenas para combinar artefatos de software de uma forma eficiente.

Capítulo 4

Combinando AspectC++ e Meta Programação Estática

Passados vários anos da introdução de Adaptadores de Cenário, a situação atual não é exatamente a esperada quando de sua introdução. O modelo Programação Orientada a Aspectos está mais difundido e novas ferramentas foram criadas com fins de suportá-lo. A implementação de AspectC++ já foi usada no desenvolvimento de sistemas operacionais [20] e mostra-se como uma alternativa adequada para o desenvolvimento de aspectos para sistemas de software básicos sobre linguagem orientada a objetos. Todavia existem considerações relevantes na aplicação de aspectos por *weaving*, sobretudo quando correlacionado ao mecanismo de *templates* de C++.

Para um *weaver* de aspectos em C++ suportar completamente a linguagem, é necessário que este seja capaz de aplicar aspectos a classes parametrizadas *(template)*. A alternativa mais óbvia de uso de um *weaver* sobre código *template* seria tratá-lo como código normal. Classes `template` e seus métodos seriam pontos de corte para a aplicação de aspectos e modificados do mesmo modo que classes comuns o são. Esta alternativa entretanto apresenta limitações. O uso de um *weaver* sobre o código de uma classe parametrizada resultaria em uma classe ainda parametrizada, mas combinada com o aspecto. Conseqüentemente todas versões desta classe parametrizada, não importando o parâmetro utilizado no *template*, sofreriam os efeitos do aspecto. Este resultado, apesar de apresentar algum grau de sucesso, pode não corresponder ao efeito desejado. Existe a possibilidade do desenvolvedor desejar aplicar o programa de aspectos apenas a uma "instância"específica da classe *template*. Por exemplo, no caso de um contêiner parametrizado com o tipo de objeto contido, o desenvolvedor pode desejar que o programa de aspecto afete apenas contêiners de números inteiros, mas não contêiners de *strings*.

```
class Abstraction{
public:

        void method_1( IF< sizeof(int)>=4, int, long long>::RET);

};
```

Figura 4.1: Exemplo de ponto de junção alterado por Metaprogramação Estática.

Outra limitação no conceito da aplicação de um *weaver* diretamente sobre código *template* é o caso da *Metaprogramação Estática*. ASPECTC++ não foi desenvolvida visando manusear código executado em tempo de compilação. A inserção ou modificação de código em um metaprograma pode causar efeitos colaterais indesejáveis. Para que um metaprograma continue válido, é necessário que nenhum código não solúvel em tempo de compilação seja adicionado a este.

Uma terceira limitação da aplicação de um *weaver* diretamente sobre código *template* diz respeito a pontos de junção sob efeito de metaprogramas. Um metaprograma pode ser usado para alterar pontos de junção tais como parâmetros em métodos. A figura 4.1 ilustra um exemplo que faz uso do metaprograma de condicional estática 3.2 para determinar o tipo a ser usado como parâmetro em um método. Caso a aplicação do programa de aspecto seja feita antes da resolução do metaprograma, o ponto de junção ainda não estaria definido. Deste modo, o desenvolvedor deseja que o metaprograma esteja resolvido antes da aplicação dos aspectos.

Devido a este tipo de limitação, a aplicação direta de programas de aspecto por meio de *weaving* sobre código *template* não é uma solução adequada. Atualmente ASPECTC++ é restrito a aplicar aspectos em códigos que fazem uso de "instâncias"de *templates*, não permitindo a aplicação de aspectos em classes parametrizadas. Metaprogramas e seus efeitos sobre pontos de junção também não são considerados por ASPECTC++. Fez-se assim necessário o estudo e desenvolvimento de uma solução para o problema do uso combinado de Metaprogramação Estática com Programação Orientada a Aspectos por *Weavers* no caso da linguagem C++.

A combinação de Programação Orientada a Aspectos por meio de *weavers* com a Metaprogramação Estática no contexto de C++ sem esbarrar nas limitações discutidas anteriormente implica na resolução de *templates* por alguma entidade, seja ela o *weaver* ou outra entidade externa. Deste modo classes parametrizadas com um tipo específico podem ser alvo da aplicação de um aspecto, bem como os efeitos de metaprogramas sobre pontos de junção seriam levados em

consideração na aplicação do aspecto. Seguindo este raciocínio, a primeira alternativa consiste na inclusão de um ambiente de avaliação e execução de metaprogramas e classes parametrizadas dentro do *weaver*. A segunda alternativa consiste em usar um pré-processador para executar eventuais metaprogramas antes do processo de composição de aspectos. Esta alternativa levaria a uma separação da resolução de Metaprogramação Estática e Programação Genérica do resto do processo de compilação.

A primeira alternativa, embutir um ambiente de execução de metaprogramas em um *weaver* de aspectos, é de relativa complexidade. Esta alternativa requer do *weaver* a capacidade de lidar com resolução e "instanciação"de *templates* com fins de manusear corretamente os pontos de junção relacionados com código metaprogramado. Resolução e "instanciação"de *templates* são algumas das tarefas mais complexas realizadas por um compilador C++ [1]. A resolução de *templates* necessita que argumentos sejam checados contra o sistema de tipos da linguagem, enquanto a resolução de *templates* depende de sofisticadas técnicas de inferência, especialmente no caso de funções *template* [1]. O tempo transcorrido do surgimento da linguagem C++ até o aparecimento dos primeiros compiladores capazes de implementar todos os recursos de *templates*, incluso especializações e funções *templates*, demonstra de forma prática a complexidade do desenvolvimento de tal ferramenta. Outro ponto desfavorável a esta alternativa é o desenvolvimento de uma funcionalidade tão complexa cujo uso estaria restrito a implementação de apenas uma ferramenta.

A segunda alternativa, executar os metaprogramas previamente ao uso do *weaver* de aspectos, libera o *weaver* de várias destas tarefas. A resolução de *templates*, bem como todas as outras tarefas relativas a execução de metaprogramas, seriam realizadas em um *pré-processador de metaprogramas*. O *weaver* só operaria em código C++ simples, lidando apenas com os resultados dos metaprogramas e classes parametrizadas. Entretanto, esta alternativa só é viável se apoiada na existência de um ambiente de compilação pré-existente, capaz de ler o programa de entrada, executar metaprogramas e gerar um programa de saída em um único nível de linguagem. Sem esta ferramenta a necessidade de desenvoler as mesmas tarefas relativas ao processamento de *templates* persistiria. Um ponto favorável desta solução é sua independência em relação a ASPECTC++, propiciando seu re-uso em qualquer situação onde seja útil a resolução adiantada dos *templates*. Isto permite que plataformas desprovidas de compiladores C++ modernos façam uso de todas as funcionalidades de Programação Genérica e Metaprogramação Estática. Esta alternativa também permite que qualquer ferramenta que opere sobre código C++ veja-se livre da necessidade de lidar

[1] Durante a resolução de funções *template*, alguns parâmetros da mesma tem seus tipos inferidos direta ou indiretamente dos argumentos da função

com código *template*.

Uma das situações onde um pré-processador de metaprogramas mostra-se uma boa alternativa é o manuseio de pontos de junção que tem como alvo potencial um metaprograma ou seção de código afetada por um metaprograma. Visto que o pré-processador é executado antes do *weaver*, o metaprograma já não existe quanto o *weaver* é usado, restando apenas os resultados de sua execução. A Metaprogramação Estática em C++ tem um estilo de codificação diferente de programas C++ tradicionais. Esta sintaxe e estilo de codificação diferentes aumentam a possibilidade da aplicação de aspectos em um nível de linguagem acarretar efeitos imprevistos sobre o outro. A adição de qualquer código que não possa ser resolvido em tempo de compilação em um metaprograma C++ pode invalidá-lo. Deste ponto de vista, restringir a aplicação de aspectos apenas aos resultados finais dos metaprogramas é perfeitamente razoável e de fato benéfico. No caso de classes parametrizadas esta restrição não se aplica, visto que classes *templates* estão no mesmo nível de linguagem que uma classe comum. Classes parametrizadas seriam transformadas em classes simples, uma classe para cada parametrização usada em uma classe *template*.

Comparando as duas alternativas, a execução do estágio de metaprogramação em separado apresenta vantagens sobre a alternativa de embutir capacidades de resolução de *templates* em um weaver de ASPECTC++. O resto do trabalho segue sobre a alternativa de contrução de um pré-processador de metaprogramas capaz de resolver Metaprogramação Estática e classes parametrizadas.

4.1 O Pré-Proprocessador de Metaprogramas

Na seção anterior foi sinalizada a preferência pelo pré-processador de metaprogramas externo ao *weaver* como uma ferramenta de suporte à combinação de Programação Orientada a Aspectos com Metaprogramação Estática e classes parametrizadas. Em resumo, fazer uso de um sistema de compilação pré-existente para implementar um pré-processador de metaprogramas parece ser uma alternativa mais prática que embutir um ambiente de execução de metaprogramas em um *weaver* de aspectos. A decisão foi também encorajada pelo fato de que o compilador C++ GNU G++ [16], um compilador de código aberto, ser bastante eficiente ao lidar com metaprogramas complexos.

De fato, a versão corrente do compilador G++ [2] já é capaz de interpretar uma uni-

[2]Os resultados providos neste trabalho são baseados em um pré-processador construído sobre a versão 3.2 do compilador G++.

dade de compilação, executar eventuais metaprogramas e prover uma saída correspondente a uma árvore sintática. Podemos instruir o compilador a fazer exatamente isso invocando o compilador G++ com o uso da opção `-fdump-tree-inlined`. Certamente a árvore sintática assim obtida não é um programa C++ e não pode ser usada como entrada para um *weaver* de aspectos. Esta árvore, porém, contém quase todas as informações necessárias para reconstrução de um programa C++ válido e equivalente ao programa de entrada original. Diferentemente do programa original, este teria os metaprogramas substituídos pelos resultados de suas execuções e cada ocorrência diferente de uma classe parametrizada seria substituída por uma classe comum. As informações que não estão presentes na saída original do compilador são na maioria pertinentes a parâmetros de funções, declarações de funções e atributos de classes que não são referenciados no código. A obtenção dessas informações é crucial para que classes e funções mantenham suas características originais [3]. Para que esta funcionalidade do compilador G++ pudesse ser usada como base para um pré-processador de metaprogramas, modificações foram introduzidas neste compilador, mais especificamente no módulo responsável pela apresentação da árvore sintática. Visto que o compilador já está de posse destas informações, é necessário apenas adicioná-las a saída do sistema de uma forma adequada. Com isto as modificações efetuadas sobre o compilador dizem respeito apenas à extração de dados e não afetam o processo de compilação propriamente dito.

Uma representação esquemática da cadeia de funcionamento do pré-processador aqui proposto é mostrada na figura 4.2. Uma árvore sintática é criada ao alimentar um compilador G++ modificado com um programa C++. Esta árvore é então submetida à seção de nosso pré-processador chamada Reconstrutor de Código, o qual produz código C++ correspondente ao programa representado pela árvore sintática. O Reconstrutor não precisa tratar de código parametrizado ou metaprogramado, visto que estes já foram resolvidos pelo compilador. O código C++ resultante pode então ser submetido a um *weaver* de aspectos, ou qualquer outra ferramenta de manuseio do código. Deste modo, pendências relativas à Metaprogramação Estática e Programação Genérica são processadas primeiro, seguidas das pendências de Programação Orientada a Aspectos e finalmente a compilação final.

[3] O tamanho de uma classe é relevante quando esta é usada como mapeamento de memória de dispositivos, que é procedimento comum em software básico. Já a manutenção de todas as funções de uma classe é importante para não remover pontos de junção de possíveis aspectos

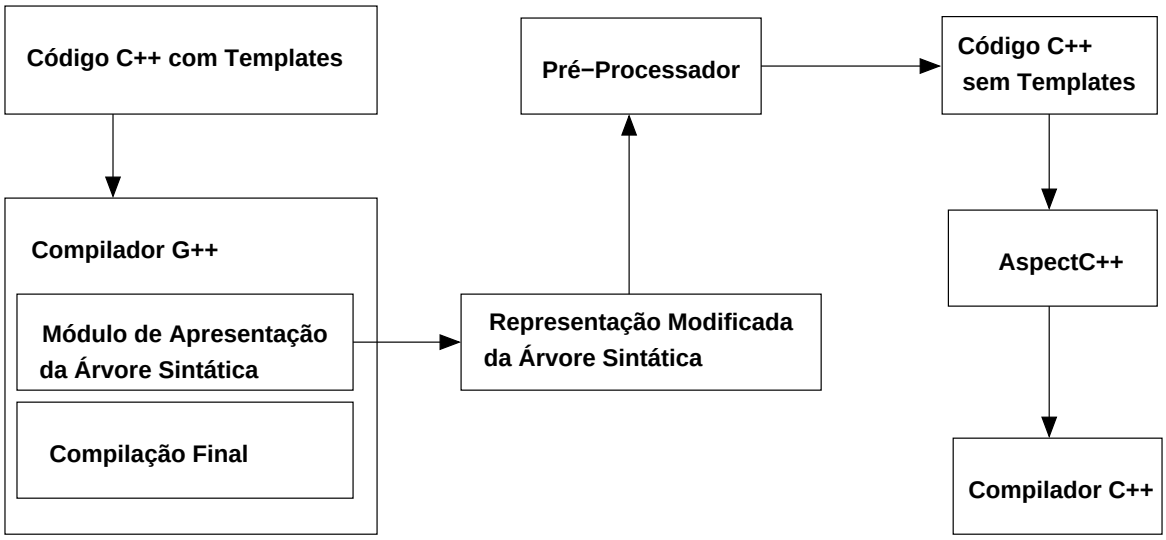

Figura 4.2: Visão geral do processo de pré-processamento.

4.2 O Reconstrutor de Código

De forma a continuar a discussão sobre a ferramenta de reconstrução de código, é importante entender os conceitos básicos dá árvore sintática provida pelo compilador G++. Para isso vamos fazer uso do exemplo clássico de metaprogramação em C++. Este exemplo consiste no fatorial resolvido estaticamente, o qual esta representado na figura 3.1. Este metaprograma é capaz de calcular o fatorial de qualquer constante inteira. Para tanto basta criar uma "instancia"do *template* com o número desejado como parâmetro.

A compilação do programa ilustrado na figura 3.1 faz com que o metaprograma `Factorial` seja executado, produzindo na saída da função `main` apenas o retorno do número `24` (o fatorial do número 4, passado como parâmetro do metaprograma). A árvore sintática provida pelo compilador G++ para este mesmo código pode ser vista na figura 4.3. Esta representação do programa pode consistir de várias árvores sintáticas, uma para cada função definida no programa. Os nodos de cada uma destas árvores representam os símbolos e tipos usados no corpo desta função, bem como parâmetros e retorno da mesma. Outros nodos representam o conteúdo de comandos, expressões e controle de fluxo de execução. A estrutura desta representação de árvore sintática é simples e pode ser usada quase que diretamente para preencher uma representação de uma nova árvore sintática dentro de um pré-processador ou compilador. Todas estas tarefas são realizadas sem a necessidade de maiores validações sintáticas e semânticas, visto que estas foram realizadas pelo compilador no passo de compilação que gerou esta árvore.

Durante o processamento da árvore sintática na figura 4.3, o reconstrutor de

```
;; Function int main() (main)
;; enabled by -dump-tree-inlined

@1     function_decl    name: @2     type: @3     srcp: factorial.cc:7
                        C            extern       body: @4
@2     identifier_node  strg: main   lngt: 4
@3     function_type    size: @5     algn: 64     retn: @6
                        prms: @7
@4     compound_stmt    line: 7      body: @8     next: @9
@5     integer_cst      type: @10    low: 64
@6     integer_type     name: @11    size: @12    algn: 32
                        prec: 32     min: @13     max: @14
@7     tree_list        valu: @15
@8     scope_stmt       line: 7      begn         clnp
                        next: @16
@9     return_stmt      line: 7      expr: @17
@10    integer_type     name: @18    size: @5     algn: 64
                        prec: 36     unsigned     min: @19
                        max: @20
@11    type_decl        name: @21    type: @6     srcp: <internal>:0
@12    integer_cst      type: @10    low: 32
@13    integer_cst      type: @6     high: -1     low: -2147483648
@14    integer_cst      type: @6     low: 2147483647
@15    void_type        name: @22    algn: 8
@16    compound_stmt    line: 7      body: @23    next: @24
@17    init_expr        type: @6     op 0: @25    op 1: @26
@18    identifier_node  strg: bit_size_type       lngt: 13
@19    integer_cst      type: @10    low: 0
@20    integer_cst      type: @10    high: 15     low: -1
@21    identifier_node  strg: int    lngt: 3
@22    type_decl        name: @27    type: @15    srcp: <internal>:0
@23    return_stmt      line: 7      expr: @28
@24    scope_stmt       line: 7      end          clnp
@25    result_decl      type: @6     scpe: @1     srcp: factorial.cc:7
                        size: @12    algn: 32
@26    integer_cst      type: @6     low: 0
@27    identifier_node  strg: void   lngt: 4
@28    init_expr        type: @6     op 0: @25    op 1: @29
@29    integer_cst      type: @6     low: 24
```

Figura 4.3: Árvore sintática produzida pelo compilador G++ para o programa `Factorial`3.1.

código gerará uma única declaração (nodo @1) para uma função(nodo @3), cujo nome é `main` (nodo @2). Esta função retorna um inteiro (nodo @6) e não recebe nenhum parâmetro (nodo @15, tipo `void`). O corpo da função (nodo @4) consiste de um comando composto cujo escopo é definido pelos nodo @8 e @24. O conteúdo do comando está definido no nodo @16, correspondendo a um comando de retorno (nodo @23). A expressão retornada (@28) é do tipo inteiro (@6) e tem o valor constante 24(@29). Fazendo uso desta árvore, o reconstrutor produziria o seguinte código.

```
int main(void) { return 24; }
```

que corresponde ao resultado da execução do metaprograma apresentado na figura 3.1.

Note-se que os nodos representantes de tipos (e.g nodo @6) são inseridos na árvore a medida em que estes vão sendo usados pelas funções. Deste modo um tipo pode ser descrito diversas vezes a partir da saída de uma única unidade de compilação, uma vez para cada função que faça uso dele. Conseqüentemente o reconstrutor de código precisa manter contabilidade dos tipos já incluídos para evitar declarar um tipo nativo da linguagem ou declarar duas ou mais vezes um tipo dentro da mesma unidade de compilação.

Apesar de não demonstrado no exemplo da figura 4.3, a árvore sintática apresentada pelo compilador G++ inclui também informações sobre os arquivos originais bem como a linha de código neste arquivo que originou tal declaração. Com o uso destas informações, é possível re-gerar as declarações nas posições originais das mesmas, desde que isto seja necessário. Como será explicada mais à frente, esta capacidade permite a utilização desta ferramenta com fins de resolver outros problemas.

As primeiras considerações surgem quando da "instanciação"de classes *template*, as quais não existiam no código original na forma de tipos completamente definidos. Quando um novo código é inserido no programa existe uma preocupação especial em inserir este código de uma forma que nenhum efeito colateral seja criado e que a semântica continue a mesma do programa original escrito com *templates*. O problema é que *templates* podem ser "instanciados"com argumentos cujos tipos são definidos em qualquer outro lugar do código. Por exemplo, uma classe parametrizada A$< typename$T$>$ pode ser concretizada com qualquer tipo de argumento que satisfaça as restrições impostas por seus métodos. Considerando que o tipo B satisfaça as eventuais restrições, a classe representada por A$<$B$>$ pode ocorrer em qualquer lugar onde o tipo B seja conhecido. Caso insiramos a declaração de uma classe correspondente a A$<$B$>$ no ponto onde o *template* fora originalmente declarado, é possível que o tipo B ainda não tenha sido declarado. Esta situação geraria um programa inválido e que não poderia ser compilado. A figura

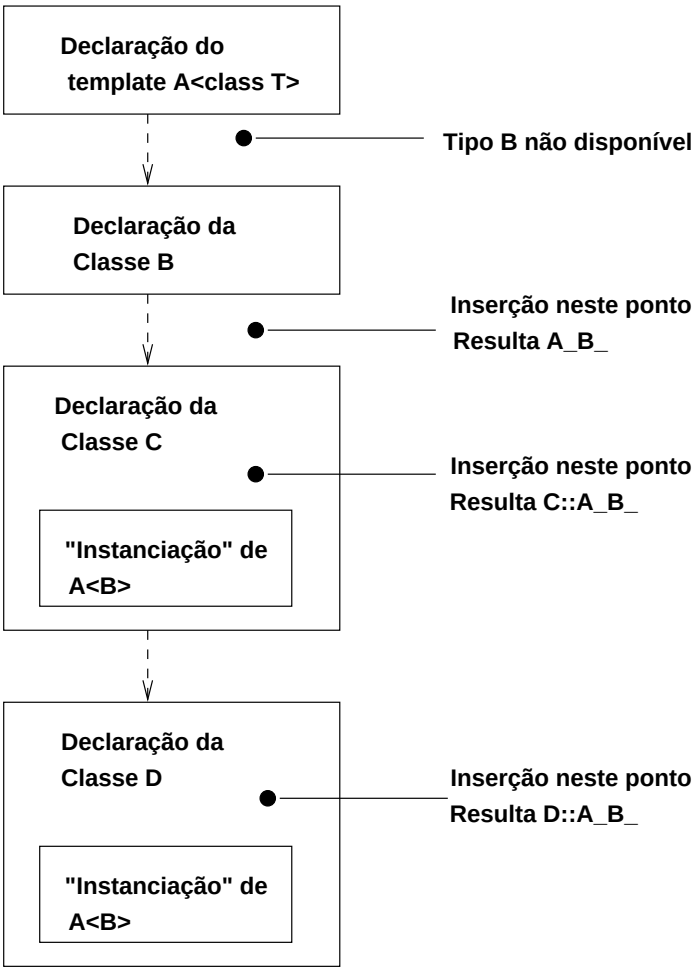

Figura 4.4: Conjunto de declarações em um programa C++ e possibilidades de inserção de novas classes.

4.4 ilustra um conjunto de declarações de classes em C++ e os possíveis pontos de inserção das classes criadas a partir da "instanciação"de *templates*.

Uma alternativa de ponto de inserção das classes é no ponto imediatamente antes da declaração original das "instancias"do *template*. Neste ponto, os tipos eventualmente usados como argumentos na classe parametrizada já estão definidos. Todavia, uma classe parametrizada pode ocorrer em diferentes escopos com o mesmo conjunto de argumentos, assim eventualmente gerando múltiplas declarações do mesmo tipo. A solução encontrada é a inserção das novas classes em um ponto onde todos os tipos usados como parâmetros do *template* já estão declarados, enquanto dentro de um escopo que possa ser alcançado por todas ocorrências similares do mesmo *template*.

Esta situação pode ser obtida imediatamente após a declaração de todos os tipos usados como parâmetros do *template*. O pré-processador mantém uma tabela de declarações de tipos em cada escopo do programa, de forma que é possível descobrir se e onde todos os tipos usa-

dos para parametrizar um *template* foram declarados. O pré-processador faz uso da contabilização da declaração de tipos para determinar qual é o ponto no código onde a nova classe pode ser inserida. A figura 4.5 ilustra esta abordagem. Caso a classe `A_B_` seja inserida imediatamente antes de seu uso, teríamos duas cópias desta classe: uma dentro da função `int foo2` e outra dentro da função `main`. Deste modo estas seriam classes diferentes e o compilador produziria o código para a função membro de ambas as classes duas vezes. O código resultante deste caso seria válido para um compilador C++, mas estaria disperdiçando recursos na forma de código replicado. Caso a classe `A_B_` seja declarada imediatamente após a declaração de `B` e deste modo uma só vez, o compilador geraria apenas uma única cópia da função `A_B_::foo`.

4.3 Integração com ASPECT-C++

O uso efetivo da técnica de pré-processamento descrita anteriormente está ligado ao suporte que pode ser dado a mesma por ASPECTC++. A preocupação mais óbvia na implantação desta técnica é a transparência do processo para qualquer um escrevendo programas de aspectos. Isto significa que o desenvolvedor do programa de aspecto não deve precisar antever os resultados do pré-processamento. As transformações realizadas sobre o código do programa não devem influir nas decisões do desenvolvedor sobre a declaração dos pontos de junção. Isto só pode ser alcançado completamente se ASPECTC++ for capaz de traduzir nomes de ocorrências de *templates* presentes em pontos de junção para o formato de nomes resultante do pré-processamento.

Enquanto ASPECTC++ não suporta este tipo de pré-processamento, o uso de regras simples de nomenclatura permite que, mesmo sem suporte direto do *weaver*, pontos de junção visando o código pré-processado possam ser facilmente escritos pelos desenvolvedores de aspectos. Alternativamente, ferramentas simples podem ser usadas para converter estes nomes. Isto significa que se o *weaver* não oferecer a possibilidade de tradução automática de nomes *templates*, como `A<int>` para o formato `A_int_`, o desenvolvedor pode facilmente escrever seus aspectos visando a nomenclatura traduzida. Espera-se que este mesmo esquema de nomenclatura simplificada facilite a implementação do suporte direto de tradução de nomes em ASPECTC++. Mesmo não resolvendo completamente o problema, estas medidas minimizam o mesmo a um nível gerenciável pelo desenvolvedor.

O pré-processador de templates mesmo sem suporte direto de ASPECTC++, pode ser usado para permitir o uso de ASPECTC++ em situações que de outra forma não seriam possíveis. A figura 4.6 ilustra um caso onde a metaprogramação estática é usada para alterar

```
// ORIGINAL PROGRAM
template<class T>
class A{
    T * o;
  public:

    int foo(){return o->foo();}
};

class B{
  public:
    int foo () { ... }
};

int foo2(){ A<B> ab1;}

int main() {
    A<B> ab2;
    return ab2.foo();
}


// PREPROCESSED PROGRAM

class B{
  public:
    void foo () { ... }
};

class A_B_ {
    public:
      B * o;

      void foo(){ return o->foo();}
};

int foo2(){ A_B_ ab1;}

int main() {
    A_B_ ab2;
    ab.foo();
    foo2();
}
```

Figura 4.5: Parametrização de classes.

a assinatura de um método. No exemplo em questão, o *weaver* de ASPECTC++ não pode aplicar o aspecto ilustrado na figura 4.7. Isto porque a assinatura do método `foo`, usado como ponto de junção do aspecto, só será definida após a execução do metaprograma `IF`. Submetendo-se este programa ao pré-processador de templates, obtêm-se como resultado o código da figura 4.8. O metaprograma usado para definir a assinatura do método já foi executado, e o código resultante é um alvo válido para ASPECTC++.

Durante o desenvolvimento deste trabalho, através de contato com a equipe desenvolvedora de ASPECTC++, verificou-se o esforço na capacitação do uso de *templates* em ASPECTC++. Atualmente desenvolve-se a capacidade de operar com aspectos programados genericamente, ou seja, aspectos *template*. Estes programas de aspectos podem receber tipos como parâmetros, permitindo um maior reuso dos mesmos. Esse trabalho, apesar de estender o potencial de uso ASPECTC++, ainda não cobre completamente a interação de ASPECTC++ com código metaprogramado e classes parametrizadas. Permitir que aspectos sejam implementados com mecanismo de programação genérica não resolve as questões relativas a pontos de junção afetados por metaprogramas, bem como não contempla o uso de aspectos em classes parametrizadas. O uso combinado de ambas as técnicas descritas aqui e das novas facilidades de ASPECTC++ são possivelmente complementares e merecedoras de um novo estudo no futuro.

4.4 Usos Secundários para o Pré-Processador

Os resultados do desenvolvimento do pré-processadores despertam a possibilidade de usos secundários da mesma tecnologia. O pré-processador de *templates* pode ter seu uso estendido a uma diversidade de problemas relacionados com *templates*. Esta técnica pode ser usada para eliminar o problema de *templates* com qualquer ferramenta de manipulação de código C++, não apenas ASPECTC++. Outro uso desta tecnologia é converter código C++ em código C++ sem *templates* para permitir que programas recentes sejam portados para plataformas antigas, nas quais a seleção de compiladores C++ é restrita. Um exemplo de tal sistema á a plataforma de rede LANAI, para qual apenas compiladores GCC da série 2.X estão disponíveis. Do mesmo modo as possibilidades de futuro desenvolvimento desta ferramenta em um conversor C++ para C mostram-se promissoras e proveitosas. Diversas plataformas de sistemas embutidos ainda não são contemplados com compiladores C++, estando restritos a compiladores C e poderiam beneficiar-se de uma ferramenta deste tipo.

Esta ferramenta é usada experimentalmente para extrair informações sobre ar-

```
template<bool COND, class EXP1, class EXP2>
struct IF{
    typedef EXP1 RET;
};

template<class EXP1, class EXP2>
struct IF<true,EXP1,EXP2>{
        typedef EXP1 RET;
};

template<class EXP1, class EXP2>
struct IF<false,EXP1,EXP2>{
        typedef EXP2 RET;
};


class Abstraction{

    int m;

public:

    void foo( IF< sizeof(int)>=4,int,long long >::RET value)
    {
        m=value;
    }
};



int main()
{
  Abstraction abs;
  abs.foo(45);
  return 1;
}
```

Figura 4.6: Código não apto à aplicação de ASPECTC++.

```
aspect SimpleAspect {

  pointcut simple_pointcut() = execution ("% Abstraction::foo(int)");

  advice   simple_pointcut() : before() { /* ... advice code... */ }

}
```

Figura 4.7: Aspecto com o método `foo` como ponto de junção.

```
class Abstraction
{
  public: void foo( int   value);
  private:  int   m;
};

int   main()
{
  Abstraction abs;
  abs.foo ( 45 ) ;
  return     1   ;
}


void Abstraction::foo( int   value)
{
  m = ( int  )( value);
}
```

Figura 4.8: Código apto à aplicação de ASPECTC++.

quivos incluídos no processo de compilação do sistema operacional *EPOS*, as quais são usadas no processo de ligação do mesmo. Isto faz desta solução um trabalho reutilizável e expansível, mesmo que ASPECTC++ venha a um dia superar sozinha suas limitações quanto a aplicação de aspectos em código *template*.

Capítulo 5

Adaptadores de Cenário versus Weavers de Aspectos

Até o capítulo anterior foi discutida uma implementação de Adaptadores de Cenário usando Metaprogramação Estática, bem como foram discutidas as dificuldades de aplicar programas de aspectos por meio de *weavers* em código *template*. Uma solução para estas dificuldades foi apresentada na forma de um pré-processador de metaprogramas e classes parametrizadas. Neste capítulo procede-se uma comparação entre as técnicas de Adaptadores de Cenário e Programação Orientada a Aspectos por meio de *weavers*, usando ASPECTC++ como *weaver*.

Apesar de compartilharem um objetivo comum, Adaptadores de Cenário e ASPECTC++ têm características e peculiaridades próprias. As diferenças entre ambas as técnicas surgem tanto na capacidade de definir pontos de junção para os aspectos, tanto como, na implementação dos aspectos.

A proposta de Adaptadores de Cenário é centrada na idéia de abstrações entrando e saindo de um cenário de operação, permitindo que ações sejam executadas durante estes momentos. Uma característica importante de Adaptadores de Cenário é a implementação ligada a um *framework* de suporte. A criação de novos aspectos fica deste modo afetada pela ligação com o *framework*. Durante a inclusão de novos aspectos no sistema, algumas construções do *framework* precisam ser manuseadas e receber código pertinente a ativação deste novo aspecto. Por outro lado, a interação com Programação Genérica [24] é simples e direta no mecanismo de Adaptadores de Cenário. Para alcançar o mesmo resultado, os *weavers* atuais precisam usar de uma ferramenta de pré-processamento, como a apresentada anteriormente, o que torna mais complexo o processo de compilação. Finalmente, a técnica de Adaptadores de Cenário não necessita de uma solução de

linguagem externa à linguagem nativa de implementação.

A abordagem por meio de *Weavers*, aqui representada por AspectC++, é capaz de usar expressões complexas para identificar vários tipos de pontos de junção dentro do código alvo. O sistema de ponto de junção provido por linguagens de programação de aspectos pode operar sobre alvos além do alcance de Adaptadores de Cenário. Um exemplo é o ponto de junção `cflow` provido por AspectC++ e que possibilita a filtragem da ativação dos aspectos de acordo com o fluxo de execução do programa. Isto permite que um aspecto seja ativado apenas quando chamado a partir de uma função específica.

Outro conjunto de diferenças pode ser visto, com relativo grau de subjetividade, na avaliação da facilidade de escrever programas de aspecto eficientes e compreensíveis. Adaptadores de Cenário fazem uso de mecanismos de Metaprogramação Estática providos por C++. Estes mecanismos de linguagem são muitas vezes conhecidos por uma sintaxe complexa e desajeitada [30], enquanto as abordagens baseadas em *weavers* contam com linguagens especialmente desenvolvidas ou adaptadas para a escrita de aspectos. A sintaxe de Metaprogramação Estática em C++ pode ser vista como uma vantagem a favor de *weavers* de aspectos no quesito facilidade de desenvolvimento. Todavia, como será discutido nos estudos de caso a seguir, o desenvolvimento de aspectos pode mostrar resultados contrários ao esperado neste quesito.

Por fim, é possível apontar vantagens da implementação de Adaptadores de Cenário aqui estudada no controle da ativação dos aspectos. A definição de que classes serão afetadas por um aspecto pode ser mantida completamente a parte do aspecto e junto a um *framework*. Os mecanismos de controle do *framework* também permitem que dependências entre aspectos sejam controladas de forma simples. Deste modo, um aspecto pode ativar outro, caso existam dependências entre eles. Não fica claro como se pode incorporar tal comportamento na aplicação de aspectos por *weavers* sem causar replicação de código entre os aspectos e sem gerar um correlacionamento excessivo entre tais aspectos.

5.1 Estudos de Caso

Visando avaliar as duas técnicas de Programação Orientada a Aspectos abordadas neste trabalho, foram desenvolvidos programas de aspecto com ambas as técnicas. Estes programas foram aplicados sobre abstrações simples em programas de teste. Os aspectos selecionados para os testes foram: Atomicidade, Compressão, Identificação e Temporização.

O aspecto denominado Atomicidade provê a funcionalidade de atomicidade ao

introduzir um mecanismo de sincronização (e.g. `Mutex`) para coordenação de sincronização de classes, objetos e do sistema como um todo. O programa de aspecto também é responsável pela ativação do mecanismo de `Mutex` nos métodos da abstração alvo quando necessário. Este aspecto, quando ativo confere ao sistema características de sincronização *(Thread-Safety)*. Este aspecto transforma a operação sob sua influência em uma seção crítica, impedindo assim a ocorrências de condições de corrida.

O aspecto Compressão provê funcionalidade de compressão (e.g. LZW, zip, etc.) de dados. O método de compressão utilizado é ortogonal ao programa de aspecto e pode ser trocado por outro conforme a necessidade.

O aspecto Identificação provê mecanismos de identificação unívoca de objetos. Esta capacidade estende-se desde simples ponteiros até identificação de objetos em sistemas distribuídos. Este aspecto engloba a identificação de objetos em um único processo, múltiplos processos ou múltiplos sistemas. O aspecto é responsável pela adição da capacidade de representar e reportar a identificação de um objeto do sistema.

O aspecto Temporização permite que um *timeout* seja estipulado para a realização de uma determinada tarefa. Caso a tarefa não tenha sido concluída ao término deste período, a mesma é abortada e o fluxo de execução é retomado no ponto de retorno da tarefa.

Todos os testes foram desenvolvidos dentro do sistema LINUX, usando o compilador GNU G++ 3.2.2 para a compilação do código final [1]. A ferramenta ASPECTC++ usada é de versão 0.8.1. Os aspectos citados e implementados fazem parte do sistema operacional *EPOS* no formato de Adaptadores de Cenário.

5.2 Atomicidade

O aspecto Atomicidade é um exemplo comum da programação orientada a aspectos. A necessidade de acesso atômico a objetos, classes ou ao sistema como um todo é comum em sistemas onde existe concorrência. Este aspecto pode ser usado em três níveis diferentes, os quais diferem no nível de paralelismo suportado. Quando o aspecto é aplicado em nível de sistema, apenas um fluxo de execução (*thread*) é capaz de executar as operações sincronizadas do sistema, promovendo nível zero de reentrância. Níveis diferentes de reentrância são suportados quando o aspecto é aplicado em nível de classes e de objetos. O primeiro caso garante-se que apenas operações realisadas sobre objetos de tipos diferentes são executáveis simultaneamente. Já

[1] O pré-processador de *templates* também é uma modificação sobre GNU G++ 3.2.2

```
class ObjectLocked{

    Mutex _mutex;
    static   Mutex _static_mutex;
public:

    void * operator new(unsigned int size)
    {
        static_enter ();
       return new char[size];
    }

    void operator delete(void* o){static_leave();}

    ObjectLocked(){ static_leave ();}

    ~ObjectLocked(){static_enter();}

    static inline void static_enter (){  _static_mutex.lock (); }

    static inline void static_leave (){  _static_mutex.unlock (); }

    inline void enter() { _mutex.lock();}

    inline void leave(){  _mutex.unlock(); }

};
```

Figura 5.1: Atomicidade em nível de objetos com Adaptadores de Cenário.

o segundo permite que quaisquer operações, desde que realizadas sobre objetos diferentes, sejam executadas simultaneamente.

Ambas as implementações deste programa de aspecto inserem objetos de uma estrutura de sincronização em suas abstrações alvo e fazem uso de seus métodos `lock` e `unlock` para sincronizar o acesso aos pontos de junção. A implementação deste aspecto é simples em ambas as técnicas, mas é feita de maneira ligeiramente diferente.

A implementação de Atomicidade em nível de objetos por meio de Adaptadores de Cenário pode ser vista na figura 5.1. O programa de aspecto provê os métodos `enter()`, `leave()`, `static_enter()` e `static_leave()` que serão chamados pelo Adaptador de Cenário. Os métodos `enter()` e `leave()` operam sobre uma "instância"de `Mutex` que é agregada pelo programa de aspecto a cada objeto da abstração sob efeito do aspecto. Os métodos `static_enter()` e `static_leave()` são métodos de classe e operam sobre um membro de classe do tipo `Mutex`. Estes dois métodos são utilizados para garantir atomicidade nas atividade-

des de alocação de objetos do tipo da abstração. A atomicidade dos procedimentos de criação de objetos é garantida com a chamada de `static_enter()` no operador `new` e pela chamada de `static_leave()` no construtor do aspecto. O procedimento de destruição de um objeto é sincronizado, chamando-se o método `static_enter()` no destrutor da classe e chamando-se o método `static_leave()` no operador `delete`.

A implementação de Atomicidade em nível de sistema, ainda por meio de Adaptadores de Cenário, pode ser vista na figura 5.2. Esta implementação funciona com um princípio semelhante à versão anterior, mas utiliza apenas um membro de classe do tipo `Mutex` para prover toda a sincronização. Esta implementação garante que toda sincronização do sistema seja feita sobre o mesmo `Mutex`. A última versão do aspecto Atomicidade por meio de Adaptadores de Cenário pode ser vista na figura 5.3. A classe `Class_Locked`, que implementa este aspecto, é implementada como uma classe *template*, garantindo que para cada abstração sob efeito do aspecto exista uma versão diferente de `Class_Locked`. Isto é necessário para que o membro de classe `_class_mutex` seja único para cada uma das abstrações, enquanto o membro `_static_mutex`, usado em `System_Locked` é comum para todas as abstrações.

A implementação do aspecto Atomicidade em ASPECTC++ foi separada em duas partes. Um programa de aspecto dá à abstração alvo as capacidades de `lock()`, `unlock()`, `static_lock()` e `static_unlock()`, bem como insere os objetos da classe de sincronização na abstração alvo. Um segundo aspecto, descrito na figura 5.7, é usado para ativar os métodos do primeiro em pontos de junção escolhidos. A razão de o aspecto ter sido desenvolvido deste modo é diminuir a replicação de código, que de outra forma seria inserido nas três versões deste aspecto.

A implementação de Atomicidade em nível de objeto pode ser vista na figura 5.4 e se assemelha em muito ao código da mesma versão do aspecto em Adaptadores de Cenário. O programa de aspecto define um ponto de junção como sendo a abstração alvo. Sobre este ponto de junção são inseridos dois membros do tipo `Mutex`, um deles sendo um membro de classe. Estes membros são usados nos métodos `enter()`, `leave()`, `static_enter()` e `static_leave()`, os quais também são inseridos pelo mesmo aspecto. Devido ao mecanismo de funcionamento de membros de classe em C++, cada um dos `Mutex` declarados como membros de classe, precisa ter seu símbolo definido através de uma variável em uma unidade de compilação. Isto é necessário para suprir os símbolos requeridos pelo processo de ligação. Os recursos providos por ASPECTC++ permitem a definição destes símbolos através do mecanismo denominado *non-inline introductions* e que pode ser visto no fim deste exemplo, na forma de um *advice* declarado fora do corpo do aspecto. Para que este mecanismo funcione é necessário que cada uma das classes

```
class SystemLocked{
protected:
    static   Mutex _static_mutex;

public:

    void * operator new(unsigned int size)
    {
        static_enter ();
        return new char[size];
    }

     void operator delete(void* o){static_leave();}

     SystemLocked(){ static_leave();}

     ~SystemLocked(){static_enter();}

    static inline void static_enter () {  _static_mutex.lock (); }

    static inline void static_leave () {  _static_mutex.unlock (); }

    void inline enter () { static_enter ();}

    void inline leave () {  static_leave (); }

};
```

Figura 5.2: Atomicidade em nível de sistema com Adaptadores de Cenário.

```
template<class Abs>
class ClassLocked {

    static   Mutex   _class_mutex;

public:

    void * operator new(unsigned int size)
    {
        static_enter ();
        return new char[size];
    }

    void operator delete(void* o){static_leave();}

    ClassLocked(){ static_leave();}

     ~ClassLocked(){static_enter();}

    inline  void enter () {  _class_mutex.lock (); }

    inline  void leave () {  _class_mutex.unlock(); }

    static  inline  void  static_enter  () {  _class_mutex.lock (); }

    static  inline  void  static_leave  () {  _class_mutex.unlock(); }

};
```

Figura 5.3: Atomicidade em nível de classe com Adaptadores de Cenário.

```
aspect Object_Locked{

        pointcut  object_locked()="Abstraction";

        advice object_locked ():  static  Mutex _static_mutex;
        advice object_locked ():  Mutex _mutex;

        advice object_locked ():
          static  void  static_lock  (){  _static_mutex.lock ();}

        advice object_locked ():
          static  void  static_unlock (){ _static_mutex.unlock();}

        advice object_locked ():  void lock () { _mutex.lock();}
        advice object_locked ():  void unlock() { _mutex.unlock();}
};

advice "Abstraction" :  Mutex Object_Locked::_static_mutex;
```

Figura 5.4: Atomicidade em nível de objetos com AspectC++.

alvo do aspecto seja listada na declaração de *non-inline introduction* e que possua ao menos um método implementado de modo não `inline`. A necessidade de repetição da expressão de ponto de corte é um empecilho menor, mas pode eventualmente gerar erros de inconsistência quando grandes conjuntos de classes são alvos dos aspectos. A necessidade de ao menos um método ser implementado de forma não *inline*, pode vir a ser um problema em alguns sistemas que possuem classes implementadas de forma completamente `inline`. Todavia esta é uma situação incomum.

A implementação de Atomicidade em nível de classe por meio de AspectC++ pode ser vista na figura 5.5 e também funciona de forma semelhante a sua correspondente em Adaptadores de Cenário. A última versão de Atomicidade em AspectC++, atomicidade em nível de sistema, é representada na figura 5.6. Esta implementação faz uso de um objeto único do tipo `Mutex` para a sincronização de todas as abstrações por ela afetadas.

Os métodos de sincronização inseridos pelas três implementações de Atomicidade são utilizados pelo programa de aspecto da figura 5.7. Este programa de aspecto define como pontos de junção todos os métodos da abstração cujos nomes são iniciados pela string *method*. Antes da execução desses métodos uma chamada a `enter()` é efetuada. Finalizada a execução dos métodos, é efetuada uma chamada ao método `leave()`. Este mesmo programa de aspecto define outros dois pontos de junção como sendo a construção e destruição de objetos do tipo da abstração. Os métodos `static_enter()` e `static_leave()` são chamados na entrada e saída destes pontos de junção, garantindo assim a atomicidade do processo de criação e destruição

```
aspect Class_Locked {

        pointcut  class_locked()="Abstraction";

        advice class_locked ():  static  Mutex _class_mutex;

        advice class_locked ():
          static  void  static_lock  () { _class_mutex.lock ();}
        advice class_locked ():
          static  void  static_unlock () { _class_mutex.unlock();}

        advice class_locked ():  void lock () { _class_mutex.lock ();}
        advice class_locked ():  void unlock(){_class_mutex.unlock();}
};

advice "Abstraction" :  Mutex Class_Locked::_class_mutex;
```

Figura 5.5: Atomicidade em nível de classes com ASPECTC++.

de objetos.

5.3 Compressão

A necessidade de compressão de dados pode ocorrer por diversos motivos, mas normalmente está relacionada a limitações na estocagem ou transmissão de dados. Este programa de aspecto executa a compressão de dados passados como parâmetros para os seus pontos de junção, repassando o resultado desta operação para o corpo do método original.

A principal diferença entre o aspecto Compressão e o aspecto Atomicidade está no tipo de atividade executada no momento em que o ponto de junção é alcançado. O aspecto Atomicidade apresenta um funcionamento baseado em uma mudança de estado durante a entrada e saída de um cenário. O aspecto Compressão opera sobre parâmetros do ponto de junção, comprimindo-os e repassando-os ao ponto de junção original antes que a execução deste seja retomada.

A implementação do aspecto Compressão com uso de Adaptadores de Cenário pode ser vista na figura 5.9. Neste aspecto o método `enter()` possui parâmetros para receber dados e o tamanho dos mesmos, bem como parâmetros para retornar os dados comprimidos e o tamanho destes. Este método "instancia"um objeto responsável pelo encapsulamento dos algoritmos de compressão e faz uso de um de seus métodos para executar a compressão de dados. Uma segunda parte do tratamento deste aspecto é a troca dos argumentos originais pelos argu-

```
static Mutex _static_mutex;

aspect System_Locked {

        pointcut system_locked()="Abstraction";

        advice system_locked():
                static void static_lock () { _static_mutex.lock();}
        advice system_locked():
                static void static_unlock () { _static_mutex.unlock();}

        advice system_locked(): void lock(){ _static_mutex.lock();}
        advice system_locked(): void unlock(){_static_mutex.unlock();}
};
```

Figura 5.6: Atomicidade em nível de sistema com ASPECTC++.

```
aspect Atomic{

        pointcut methods()="% Abstraction::method%(...)";

        advice execution(methods()):
                before() { tjp->target()->lock();}
        advice execution(methods()):
                after () { tjp->target()->unlock();}

        advice construction("Abstraction"):
                before() { tjp->target()->static_lock();}
        advice construction("Abstraction"):
                after () { tjp->target()->static_unlock();}
        advice destruction("Abstraction"):
                before() { tjp->target()->static_lock();}
        advice destruction("Abstraction"):
                after () { tjp->target()->static_unlock();}
};
```

Figura 5.7: Ativação dos métodos de sincronização em ASPECTC++.

```
template<typename Abs, typename Aspect>
class Adapter : public Abs, public Aspect{
public:

inline void write(const char *data, int size)
   {
        char compressed_buff[size];
        int  compressed_size;
        Aspect::enter(data,compressed_buff,size,compressed_size);
        Abs::write(compressed_buff,compressed_size);
        Aspect::leave();
   }
}
```

Figura 5.8: Substituição de argumentos no `Adapter`.

mentos obtidos com o processo de compressão, antes que o método original da abstração possa ser executado. Esta substituição é feita no `Adapter` e pode ser vista na figura 5.8. Um adaptador simples é usado para aplicar um único aspecto a uma abstração na chamada de um método `write`, que deve realizar uma operação de I/O com os dados recebidos no parâmetro `data`. O método que encapsula a chamada ao método da abstração realiza a troca de argumentos. A troca é feita criando-se um *buffer* onde os dados comprimidos podem ser colocados, bem como uma variável para guardar o tamanho do novo conjunto de dados. O método `enter()` do aspecto é chamado com estes argumentos junto com os argumentos originais da função. Após a execução de `enter()` o método da abstração é chamado com o *buffer* e a variável `compressed_size` como argumentos. Desta forma a substituição de argumentos na chamada de métodos é realizada de forma simples por Adaptadores de Cenário.

Implementar o mesmo aspecto em ASPECTC++ poderia levar à confecção de diferentes códigos. Uma abordagem que pode ser considerada natural e de uso provável na primeira tentativa de compor este aspecto seria o código da figura 5.10. Este código é similar ao conceito empregado no exemplo de Adaptadores de Cenário. A idéia de que uma área de armazenamento temporária é reservada para operar os dados comprimidos é mantida. Este *buffer* é usado como argumento na chamada do método da abstração. Infelizmente esta implementação não é correta e esconde uma armadilha na manipulação de argumentos: os parâmetros declarados e usados no aspecto podem aparentar serem os mesmos parâmetros do ponto de junção propriamente dito; entretanto são apenas correlacionados durante o processo de *weaving*. Os argumentos recebidos por estes parâmetros constituem apenas cópias dos argumentos do método original. Conseqüentemente, operações realizadas sobre os argumentos dentro de um aspecto não têm efeito sobre os

```
class Compression{

public:

    static inline void static_enter (){}
    static inline void static_leave (){}

    inline void enter(const char* in, char *out, int size_in , int &size_out)
    {
        Compressor compressor;
        compressor.compress(in, size_in,out, size_out );
    }

    inline void leave(){}

};
```

Figura 5.9: Aspecto de Compressão em Adaptadores de Cenário.

argumentos passados para método da abstração quando o fluxo normal de execução é retomado. A implementação apresentada de fato realiza a compressão dentro da seção de código do aspecto, porém esta não é repassada como argumento para o método da abstração. Desenvolvedores mais experientes de AspectC++ possivelmente seriam menos suscetíveis a realizar este tipo de erros. Mesmo assim, a declaração de parâmetros com os mesmos nomes dos parâmetros do ponto de junção é lógica e compreensível, visto que aumenta a clareza do código e, portanto, uma prática que pode ser esperada de muitos desenvolvedores. A incapacidade da ferramenta de emitir mensagens de erro quanto a este uso equivocado dos parâmetros gera uma situação em que erros podem ser facilmente criados, sendo esses erros de difícil detecção.

Uma das possíveis implementações corretas do aspecto de compressão é apresentada na listagem 5.11. Fundamentalmente esta implementação é igual à implementação anterior em tudo, menos na manipulação dos parâmetros do ponto de junção. Como pode ser visto, modificar o valor dos parâmetros do método `write()` exige uma sintaxe de pouca elegância, difícil de ser lida e de codificação sujeita a erros. Programadores acostumados com a linguagem C++ sabem que o tipo de código usado para manipular parâmetros neste exemplo é uma fonte potencial de problemas e de erros de desenvolvimento.

Enquanto a substituição de argumentos na chamada de métodos pode ser feita de forma simples em Adaptadores de Cenário, em AspectC++ a solução é de difícil compreensão e de uma codificação propensa a erros. Um problema possivelmente mais sério é a possibilidade de uma codificação aparentemente correta e natural para programadores C++ resultar em um código

```
aspect Compression{

        pointcut  compress(char *data, int size)
                =execution("% Abstraction::write (...) ")
                && args(data,size);

        advice compress(data,size): before(char *data, int  size)
        {
                Compressor compressor;
                char buffer[size];
                int  out_size;
                compressor.compress(data,size,buffer,out_size);
                data=out;
                size=out_size;
        }
};
```

Figura 5.10: Uso incorreto de substituição de argumentos em ASPECTC++.

```
aspect Compression{

        pointcut  compress(char* data, int size)=
                execution("% Abstraction::write (...) ")
                && args(data,size);

        advice   compress(data,size): before(char* data, int  size)
        {
                Compressor compressor;
                char buffer[size];
                int  out_size;
                compressor.compress(data,size,buffer,out_size);
                *((char**)(tjp->arg(0)))=buffer;
                *((int*)( tjp->arg(1)))=out_size;
        }
};
```

Figura 5.11: Aspecto de Compressão em ASPECTC++.

que não faz efeito algum sobre o programa alvo. Os erros provenientes deste tipo de engano são de difícil detecção, pois somente podem ser encontrados por experimentação ou pela análise do código gerado pelo *weaver*. Visto que o código resultante do processo de *weaving* não é confeccionado com intenções de ser lido por seres humanos, esta não é uma solução recomendável.

5.4 Identificação

A identificação unívoca de um objeto dentro de um sistema é uma idéia comum na Programação Orientada a Objetos. Usualmente a identificação unívoca de um objeto é provida de algum modo pela própria linguagem de programação. A identificação suprida pela linguagem de programação é muitas vezes um reflexo direto da posição do objeto na memória do sistema. Por exemplo, o endereço de memória em que o objeto está alocado é a identificação unívoca de um objeto em C++. Este nível de identificação é suficiente quando o objeto somente é relevante dentro de um único processo ou ambiente de execução. Mas sistemas que contem com comunicação entre objetos em processos ou ambientes de execução diferentes necessitam de um sistema de identificação mais complexo.

O aspecto Identificação implementado por meio de Adaptadores de Cenário existe em três formatos: `Pointer, Local_Id` e `Global_Id`. A implementação de `Pointer` pode ser vista na figura 5.12 e baseia-se no endereço do objeto em memória como meio de identificação. A obtenção do endereço do próprio objeto em memória é feita com o uso do identificador reservado `this`. A razão do uso deste aspecto em detrimento do uso direto de um ponteiro é a obtenção de uma interface uniforme entre todos os casos de identificação de objetos.

A identificação por meio de endereço de memória só é válida dentro de um mesmo espaço de endereçamento e, portanto, `Pointer` não pode prover identificação inter-processos do sistema. A versão do aspecto Identificação que provê esta funcionalidade é `Local_Id`, cujo código pode ser visto em 5.13. O programa de aspecto `Local_Id` provê identificação através de um par `(type,unit)`. Este par identifica a abstração da qual o objeto é uma "instância", bem como qual dentre as "instâncias"desta abstração. A contagem de "instâncias"de uma determinada classe é mantida por meio de um membro de classe `_static_unit_count`, o qual é usado para atribuir o valor de `unit` na criação de um objeto. Este aspecto é implementado como uma classe *template* para que cada abstração alvo do aspecto conte com uma versão diferente do aspecto. Isto é necessário para garantir que a contagem de "instâncias"seja única para cada abstração do sistema. O aspecto Identificação apresenta uma implementação vazia dos métodos `enter()`, `leave()`,

```
class Pointer
{
public:
    Pointer(){}

    Pointer* id () const {return (Pointer*) this;}

    bool valid () const { return (bool) this ; }

    inline void enter(){}
    static inline void static_enter (){}
    inline void leave(){}
    static inline void static_leave (){}
};
```

Figura 5.12: Identificação por Ponteiro em Adaptadores de Cenário.

`static_enter()` e `static_leave()`, visto que nenhuma ação precisa ser tomada na entrada e na saída dos métodos da abstração.

No caso de um sistema distribuído em mais de um ambiente de execução (*Host*), é necessário identificar a qual sistema o objeto pertence. Na implementação de `Global_Id`, a identificação é expandida para uma tupla *(host, type, unit)*. Este aspecto herda a capacidade de identificação inter-processsos de `Local_Id` e adiciona a esta a identificação do sistema. A implementação de `Global_Id` também herda de uma classe `LocalHost`, que provê um membro de classe para a identificação do sistema, o qual é compartilhado por todas as versões do *template* `Global_Id`. Do mesmo modo que nas outras versões deste aspecto, nenhuma ação é realizada nos métodos `enter()`, `leave()`, `static_enter()` e `static_leave()`.

A implementação de um programa de aspecto deste tipo em AspectC++ é bastante semelhante ao que é feito em Adaptadores de Cenário. A figura 5.15 contém o código da implementação de `Pointer`, que é responsável pela identificação por meio do endereço do objeto em memória. Esta implementação é simples e quase idêntica à implementação de `Pointer` por meio de Adaptadores de Cenário.

A identificação inter-processos é implementada pelo programa de aspecto denominado `Local_Id`, que pode ser visto na figura 5.16. No caso de Adaptadores de Cenário, o próprio aspecto é a representação da tupla de dados usadas para identificação do objeto, sendo isto possível devido ao mecanismo de herança usado para combinar aspectos e abstrações. Esta relação hierárquica permite que o método `id` retorne o próprio objeto através de uma conversão de tipos. No caso da implementação por meio de AspectC++ a tupla de identificação foi definida como

```
template<class Abs>
class Local_Id{

    Type_Id _type;
    Unit_Id  _unit;
    static  Type_Id  _static_unit_count ;

public:

    Local_Id ():
        _type( Traits<Abs>::class_id),
        _unit ( _static_unit_count )
    {
        _static_unit_count ++;
    }

    bool operator==(const Local_Id& l)const
    {
        return (_type==l.type() && _unit==l.unit ());
    }

    const Local_Id& id() const {return *this;}

    void id(const Type_Id &t, const Unit_Id &u){ _type=t; _unit=u;}

    bool valid () const {return (_unit!=NO_UNIT);}

    const Type_Id& type() const{return _type;}
    const Unit_Id& unit () const{return _unit;}

    inline void enter(){}
    static inline void static_enter (){}
    void inline leave(){}
    static inline void static_leave (){}
};
```

Figura 5.13: Identificação Inter-Processos em Adaptadores de Cenário.

```
class LocalHost{
protected:
    static Host_Id _local_host;
public:
    static void localHost(const Host_Id &h){_local_host=h;}
};

template<class Abs>
class Global_Id: public LocalHost, public Local_Id<Abs>{

    Host_Id _host;

public:

    Global_Id(): Local_Id<Abs>(),_host(LocalHost::_local_host){}

    bool operator==(const Global_Id& l)const
    {
        return (Local_Id<Abs>::type()==l.type()
                && Local_Id<Abs>::unit()==l.unit()
                && _host==l.host());
    }

    const Global_Id& id() const {return *this;}

    const Host_Id& host() const {return _host;}

    void id(const Host_Id& h,const Type_Id &t, const Unit_Id &u)
    {
        _host=h;Local_Id<Abs>::id(t,u);
    }

    inline void enter(){}
    static inline void static_enter (){}
    inline void leave(){}
    static inline void static_leave (){}
};
```

Figura 5.14: Identificação Remota em Adaptadores de Cenário.

```
aspect Pointer{
        pointcut pointer()="Abstraction";

public:
        advice pointer (): const void* id() const {return this;}
        advice pointer (): bool valid () const {return (bool)this;}
};
```

Figura 5.15: Identificação por Ponteiro em AspectC++.

uma classe `UnitType`.

O programa de aspecto `Local_Id` define como ponto de junção uma abstração e adiciona a esta um membro do tipo `UnitType`, um contador do número de "instâncias"da abstração como membro de classe e uma constante de identificação para a abstração. Os métodos adicionados por este programa de aspecto são semelhantes aos métodos adicionados pela versão em Adaptadores de Cenário. A atribuição de valores para a tupla de identificação é feita por meio do contador de referências da abstração, também de modo similar a como é feito em Adaptadores de Cenário.

A versão deste aspecto referente a identificação em sistemas distribuídos pode ser vista na figura 5.17. A implementação de `Global_Id` é semelhante à implementação de `Local_Id`, estendendo a tupla `(unit,type)` para `(unit,type,host)`. A identificação do sistema no qual o objeto é criado é provida de forma acessível a todas "instâncias"da abstração por meio do identificador `local_host`. Deste modo as três versões do aspecto Identificação em AspectC++ não apresentam grandes diferenças das implementações por meio de Adaptadores de Cenário.

Este estudo de caso apresentou resultados semelhantes entre Adaptadores de Cenário e AspectC++. A inclusão simples de membros e métodos a uma classe é uma tarefa que pode ser realizada sem dificuldades por ambas as técnicas.

5.5 Temporização

No ambiente de sistemas dedicados são comuns as aplicações nas quais um *time-out* é imposto à execução de determinadas tarefas. Por exemplo, para impor restrições de operação em tempo real a um sistema pode necessitar garantir os limites de tempo de execução de determinadas tarefas. Esta característica é ortogonal à tarefa realizada pela abstração do sistema e é uma candidata a implementação por meio de Programação Orientada a Aspectos.

Apesar de identificável como um aspecto, a funcionalidade de temporização não é de implementação trivial como tal. Na teoria a implementação do aspecto de temporização pode ser separada em duas etapas. O primeiro passo é a ativação de um temporizador *(Timer)* no início da execução da tarefa. Este temporizador é programado para disparar quando do término do limite de tempo para execução da tarefa alvo. A implementação desta parte do aspecto é facilmente realizada pelas duas técnicas em estudo. A chamada de um método na entrada de um cenário ou ponto de junção é uma funcionalidade básica em ambos os modelos de aplicação de aspectos e

```
class UnitType{

        Type_Id _type;
        Unit_Id  _unit;
public:
        const Type_Id& type() const {return _type;}
        const Unit_Id& unit()  const {return _unit;}
        void set(Type_Id t,  Unit_Id  u){_type=t; _unit=u;}

        bool operator==(const UnitType& o)const
        {
                return (_type==o.type() && _unit==o.unit());
        }
};


aspect Local_Id{

        pointcut  local()="Abstraction";

        advice local () :  UnitType  _local_id ;
        advice local () :  static  unsigned int  _static_unit_count ;
        advice local () :  static  const unsigned int _type_code=1;
        advice local () :  void setLocalId()
        {
                _local_id .set(_type_code, _static_unit_count );
        }
public:
        advice local ():
          const UnitType& id() const {return _local_id;}

        advice local ():
          bool valid ()  const {return (_local_id . unit ()!= NO_UNIT);}

        advice construction("Abstraction"): after ()
        {
                tjp->target()->setLocalId();
                tjp->target()->_static_unit_count++;
        }

};

advice "Abstraction" :  unsigned int Local_Id:: _static_unit_count =0;
```

Figura 5.16: Identificação Inter-Processos em ASPECTC++.

```
static Host_Id local_host;

class UnitTypeHost{

        Type_Id _type;
        Unit_Id  _unit;
        Host_Id _host
public:
        const Type_Id& type() const {return _type;}
        const Unit_Id& unit () const {return _unit;}
        const Host_Id& host() const {return _host;}
        void set(Type_Id t, Unit_Id u, Host_Id h)
        {
                _type=t; _unit=u;_host=h;
        }

        bool operator==(const UnitType& o)const
        {
                return (_type==o.type() && _unit==o.unit ());
        }
};

aspect Global_Id{

        pointcut global()="Abstraction";

        advice global () : UnitTypeHost _global_id;
        advice global () : static unsigned int _static_unit_count;
        advice global () : static const unsigned int _type_code=1;
        advice global () : void setGlobalId()
        {
                _global_id.set(_type_code, _static_unit_count, local_host);
        }
public:
        advice global ():
          const UnitTypeHost& id() const{ return _global_id;}

        advice global ():
          bool valid () const{ return (_global_id.unit ()!=NO_UNIT);}

        advice construction("Abstraction"): after ()
        {
                tjp->target()->setGlobalId();
                tjp->target()->_static_unit_count++;
        }
};

advice "Abstraction" : unsigned int Global_Id:: _static_unit_count =0;
```

Figura 5.17: Identificação Remota em ASPECTC++.

pode ser usada para ativar um temporizador.

A segunda parte do aspecto é a que apresenta maiores dificuldades. Ocorrido o disparo do alarme, ações devem ser realizadas para interromper a execução da tarefa e retornar ao fluxo de execução principal do sistema. O fluxo de execução do programa ou sistema deve ser retomado no ponto imediatamente posterior à chamada da tarefa temporizada. Estas não são ações simples de serem implementadas com ferramentas usuais de programação. O programa de aspecto precisa saber para que ponto ele deve redirecionar o fluxo de execução, desfazer a pilha do método em execução de forma correta e retomar o contexto apropriado do método que originou a chamada da tarefa interrompida. A utilização do mecanismo de tratamento de exceções da linguagem poderia ser usado para facilitar a resolução deste problema, mas as restrições de desenvolvimento para sistemas embutidos não raramente impedem o uso do mesmo. Como resultado, essa alternativa é descartada como uma solução aceitável para este problema.

Ambas as técnicas de aplicação de aspectos aqui estudadas foram projetadas para inserir chamadas de métodos em determinados pontos do código. Esta capacidade não resolve o problema de retornar o fluxo de execução ao seu ponto correto, pois não é desejável realizar uma nova chamada de função. Chamar uma nova função apenas acarretaria na entrada de um novo contexto de função, enquanto que o comportamento desejado é a saída do contexto atual. A solução deste problema exige um meio diferente de controle do fluxo de execução. A mudança do fluxo de execução para um ponto determinado pode ser feita através de código *assembly*, o qual pode ser usado em programas na linguagem C++. Para que este código tenha o efeito desejado, é necessária a existência de um símbolo conhecido na saída da função temporizada. Este símbolo pode ser usado para determinar o ponto para onde a execução deve ser desviada. A inserção de um símbolo deste tipo pode ser improvisada inserindo-se código nos métodos `leave()` e `after()` de Adaptadores de Cenário e AspectC++, respectivamente. O pseudo-código da figura 5.18 indica onde isto seria realizado.

O desempilhamento de parâmetros e a recuperação de contexto devem ser feitos ao sair da função interrompida. A determinação de como este desempilhamento deve ser realizado é dependente do compilador utilizado, o que é um problema que por si só dificulta a implementação manual desta tarefa. Um segundo problema surge quando se leva em consideração a capacidade de implementação de métodos de forma `inline` na linguagem C++. Pode ser impossível, antes do processo de compilação, determinar se o método chamado será de fato executado como um procedimento ou terá seu conteúdo inserido diretamente na função que o chama. Um método *inlined* não precisa ser desempilhado, nem possui um ponto de retorno, de modo que aplicar o desempi-

```
class Timed{

public:
        void  static_enter (){}

        void  static_leave (){}

        void enter()
        {
            Timer timer;
            timer.time(100);
        }

        void leave()
        {
           // ASM CODE

           // TARGET SYMBOL INCLUDED HERE
        }
};
```

Figura 5.18: Pseudo código para o aspecto Temporização.

lhamento do contexto de forma forçada em qualquer caso pode causar problemas. Nenhuma das técnicas de aplicação de aspectos tem mecanismos para lidar diretamente com o processo de empilhamento de contexto e parâmetros, nem como obter informações sobre como esta função será implementada pelo compilador. Esta limitação é natural, visto que ambas as técnicas são aplicadas em um momento anterior ao processo de geração de código. Ocorre que algumas informações necessárias para a correta execução deste programa de aspecto estão presentes apenas no momento em que o código final é gerado pelo compilador.

Analisando as dificuldades encontradas neste problema, e em vista da falta de uma solução robusta por meio de qualquer uma das técnicas sob estudo, é possível concluir que ambas as técnicas não mostraram-se adequadas para implementar este estudo de caso. Infere-se que este é um exemplo de aspecto cujo tratamento deve preferencialmente ser feito por uma técnica integrada ao compilador, visto que este possui as informações necessárias sobre a implementação da função a ser temporizada.

A ferramenta de reconstrução de código apresentada neste trabalho poderia, com algumas modificações, prover os mecanismos necessários para a aplicação do aspecto Temporização. No estágio de compilação do qual a árvore sintática é obtida para uso pelo reconstrutor de código, informações úteis para aplicação deste aspecto podem também ser obtidas. Caso a compilação

tenha sido executada com a opção `-finline-functions` ativada, a determinação de quais funções serão ou não *inlined* já foi realizada. Isto permite superar uma das dificuldades para a aplicação deste aspecto por meio de Adaptadores de Cenário ou AspectC++. A capacidade da ferramenta não é limitada à obtenção de informações, podendo a mesma modificar o código quando necessário. Esta capacidade permite inserção automática de um símbolo para marcação do ponto de saída da função. Testes experimentais sobre esta funcionalidade estão em fase de implementação.

5.6 Análise dos Resultados

Através dos estudos de caso apresentados, foi possível identificar pontos fortes e fracos em ambas as técnicas. Durante os estudos de caso, ambas as técnicas apresentaram resultados semelhantes no desenvolvimento de aspectos, com exceção do aspecto Compressão, em que Adaptadores de Cenário mostraram vantagem. No caso do aspecto Temporização nenhuma das técnicas foi considerada apropriada para a implementação e aplicação do programa de aspecto sem o auxílio de outras ferramentas externas. Nos casos estudados, nenhuma das técnicas demonstrou completa superioridade em relação à outra.

Uma vantagem teórica dos *weavers*, mas que não demonstrou ser tão vantajosa como o esperado, foi a extensa seleção de pontos de junção possíveis para aplicação de aspectos que é disponibilizada pelas linguagens de Programação Orientada a Aspectos. Os pontos de junção de troca de mensagens e a inserção de membros em classes são suficientes para cumprir com a grande maioria dos aspectos que venham a surgir em um sistema Orientado a Objetos. O próprio funcionamento baseado na troca de mensagens e herança torna o uso destes pontos de junção de fácil compreensão e confecção no desenvolvimento de programas de aspecto para sistemas Orientados a Objetos. O estudo de caso sobre o aspecto Temporização, em que os pontos de junção suportados por Adaptadores de Cenário não foram suficientes, também não pôde ser resolvido de forma satisfatória por meio de AspectC++. Deste modo, verificou-se que o sistema de definição de pontos de junção de AspectC++ não é motivo suficiente para declarar superioridade desta técnica.

Uma implementação de Adaptadores de Cenário ambientada em um *framework*, como discutido anteriormente, pode ser vantajosa. A seleção de que abstrações serão afetadas e por quais versões de cada um dos aspectos pode ser feita de forma eficiente por meio de metaprogramas alimentados de informações provenientes de um *framework*. Em AspectC++ a determinação de

	Object_Locked	Class_Locked	System_Locked
AspectC++	5824 bytes	5843 bytes	5841 bytes
Adaptadores de Cenário	5491 bytes	5884 bytes	5507 bytes

Figura 5.19: Resultado da aplicação de Atomicidade em Estudo de Caso.

que abstrações são afetadas por um aspecto é feita de forma explícita no próprio programa de aspecto. Deste modo, um aspecto pode ser implementado por meio de Adaptadores de Cenário de forma mais independente da abstração alvo do que por meio de ASPECTC++.

5.6.1 Eficiência de Implementação

Para efetuar uma avaliação da eficiência de Adaptadores de Cenário e ASPECTC++, os aspectos dos estudos de caso foram aplicados sobre programas teste e o tamanho do código resultante foi mensurado. Os testes foram todos compilados com o compilador GNU G++ com o uso da *flag* -O3 de seleção de otimizações. Esta selação de otimizações ativa a maior parte das otimizações do compilador visando desempenho. Os programas foram compilados como aplicativos de modo texto em um sistema operacional LINUX em uma máquina de arquitetura IA32. Uma mesma abstração foi usada para testar todos os programas de aspectos. Esta abstração apresentava como interface métodos para acessar membros da mesma e um par de métodos capazes de escrever e ler um *buffer* de dados em um arquivo.

Os testes sobre o aspecto Atomicidade incluíram a criação de um objeto do tipo `Abstraction`, seguido de acesso tanto para recuperação como para determinação de valores para membros do objeto. No fim do teste o objeto é desalocado. O aspecto Atomicidade teve cinco (5) pontos de junção neste teste, incluindo criação e destruição de objetos. Este teste foi realizado com as três versões do aspecto Atomicidade e os resultados podem ser vistos na figura 5.19.

O teste do aspecto Compressão é feito com a leitura de um arquivo de dados e posterior escrita destes dados em um outro arquivo por uma “instância”da abstração. Este aspecto teve apenas um ponto de junção no método de escrita dos dados para o aqruivo. Os resultados deste teste podem ser vistos na figura 5.20.

O aspecto Identificação foi testado pela criação de objeto do tipo da abstração, seguida pela requisição e comparação das identificações destes objetos. Este aspecto teve um ponto de junção na forma da classe `Abstraction`, que recebeu as capacidades de identificação

	Compressed
AspectC++	12974 bytes
Adaptadores de Cenário	12791 bytes

Figura 5.20: Resultado da aplicação de Compressão em Estudo de Caso.

	Pointer	Local_Id	Global_Id
AspectC++	5645 bytes	5801 bytes	5857 bytes
Adaptadores de Cenário	5531 bytes	5668 bytes	5700 bytes

Figura 5.21: Resultado da aplicação de Identificação em Estudo de Caso.

por meio de `Pointer, Local_Id` e `Global_Id`. Os resultados deste teste podem ser vistos na figura 5.21.

Devido à dificuldade na implementação do aspecto Temporização tanto por meio de Adaptadores de Cenário como por meio de AspectC++, não foi realizado nenhum tipo de mensuração sobre este estudo de caso.

Os resultados indicam melhor desempenho na aplicação de aspectos por meio de Adaptadores de Cenário. Com exceção de Atomicidade em nível de classes, onde AspectC++ teve melhor resultado, Adaptadores de Cenário consistentemente produziram código menor que AspectC++. Estes resultados indicam que o código *weaved* por AspectC++ não é tão facilmente otimizável pelo compilador quanto é o código produzido por Adaptadores de Cenário.

Além de mensurar o tamanho do executável gerado por ambas as técnicas, foram realisados testes de performance para as implementações dos estudos de caso Atomicidade e Compressão. O teste consistiu em executar 400 mil iterações do conjunto de pontos de junção já usados no teste anterioremente descrito. O tempo de execução foi medido com auxílio da função `gettimeofday()` do sistema Linux. Para minimizar os efeitos do sistema de escalonamento de processos do sistema operacional, o laço de teste inicia apenas após o retorno de uma função de sistema `sleep()`.

Com fins de evitar que a execução do algorítmo de compressão camuflasse os resultados, o teste de perfomance do aspecto Compressão foi realizado com uma abstração de compressão falsa. Esta abstração possui a mesma interface da abstração de compressão usada no teste de tamanho do executável, mas não realiza a compressão de dados propriamente dita.

	Atomic Object	Atomic Class	Atomic System
AspectC++	1333 microsegundos	1331 microsegundos	1333 microsegundos
Adaptadores de Cenário	1112 microsegundos	1111 microsegundos	1112 microsegundos

Figura 5.22: Resultado das medições de performance para Atomicidade.

	Compressed
AspectC++	2590 microsegundos
Adaptadores de Cenário	1163 microsegundos

Figura 5.23: Resultado das medições de performance para Compressão.

Os resultados dos testes podem ser vistos nas figuras 5.22 e 5.23. Os resultados não apresentam surpresas e mantêm a tendência de um melhor resultado quando utilizando Adaptadores de Cenário. Apesar de ambos aspectos realizarem as mesmas tarefas, novamente Adaptadores de Cenário produrizam um código mais facilmente otimizável pelo compilador utilizado.

5.6.2 Clareza e Facilidade de Implementação

Nos quesitos complexidade de implementação e clareza de codificação, a análise corre o risco de tornar-se subjetiva. Na tentativa de racionalizar esta análise, seguem várias observações sobre ambas as técnicas.

Grande parte do trabalho de desenvolvimento com Adaptadores de Cenário consiste em implementá-los com métodos de encapsulamento para todos os possíveis pontos de junção das abstrações do sistema. Implementações mais complexas, como a utilizada no sistema EPOS, implicam na extensão deste trabalho para outros artefatos de software envolvidos no processo de adaptação. Para que o uso de adaptadores seja transparente para o desenvolvedor de aplicações é necessário o uso de algum mecanismo de definição de pseudônimos para tipos, ou outro mecanismo de efeito similar. A criação deste mecanismo também é parte dos custos de uso de Adaptadores de Cenário.

O desenvolvimento de aspectos em ASPECTC++ possui sua complexidade bastante concentrada na linguagem, não apresentando uma estrutura única ou conceito concentrador da complexidade fora da mesma. Todavia, o aprendizado de uma linguagem nova para descrição

de aspectos deve ser considerado como um acréscimo de complexidade. Este aprendizado é importante devido ao fato de algumas características da linguagem serem possivelmente enganosas para programadores de linguagens "tradicionais", como foi discutido no estudo do aspecto Compressão. Adaptadores de Cenário fazem uso apenas de ferramentas da própria linguagem de programação do sistema, não exigindo que o desenvolvedor divida sua atenção entre duas linguagens.

A técnica de Adaptadores de Cenário traz vantagens na facilidade de substituição de parâmetros de função, quando comparada à aplicação de aspectos por meio de *weavers*. A alternativa aparentemente mais simples de implementação da substituição de um parâmetro em ASPECTC++ vem a ser um código que não executa a tarefa desejada, mas que é correto segundo o processo de compilação. Este tipo de ocorrência pode gerar erros de difícil detecção. É da opinião do autor que a necessidade de referenciamento direto sobre o ponto de junção na forma de um objeto e por meio de uma API específica, torna o código menos claro que seu equivalente por *wrapping* como é feito em Adaptadores de Cenário. Esta necessidade torna complexa a confecção de aspectos que de outra forma seriam simples.

Um ponto importante no desenvolvimento de programas é a correção e detecção de erros. Em ambas as técnicas se encontram problemas quanto a estes quesitos. No caso da linguagem ASPECTC++ existem dificuldades na visualização do resultado final da combinação do código do programa de aspecto e da abstração alvo. É difícil prever efeitos colaterais do *weaver*, tais como a possível inclusão de código em métodos que originalmente seriam eliminados na passagem pelo compilador, assim aumentando desnecessariamente o tamanho do programa resultante. A análise do produto do processo de *weaving* também não é uma tarefa trivial, visto que este código não é confeccionado para ser lido por seres humanos. No caso de Adaptadores de Cenário, a presença de código metaprogramado tende a gerar mensagens de erros confusas na ocorrência de uma falha no processo de compilação. A figura 5.24 mostra as mensagens de erro provenientes do uso de um argumento errado em um metaprograma de um adaptador no arquivo `adapter.h`. Note-se que não existe nenhuma indicação do fato do erro ter sido causado por um argumento inválido. Um exemplo de código produzido por APECTC++ pode ser visto na figura 5.25. Este código é apenas uma fração do código produzido pelo *weaver* de ASPECTC++ na aplicação do aspecto Atomicidade em nível de objetos. O código completo gerado para este aspecto alcança 389 linhas, tornando difícil eventuais análises do mesmo.

```
abstraction.h: In static member function 'static void FOR_ALL<TYPE_LIST, DO>::EXEC(T*)
               [with T = Adapter<Abstraction, main()::AspectList>, TYPE_LIST =
               Abstraction, DO = Adapter<Abstraction, main()::AspectList>::DO_ENTER]
adapter.h:29: instantiated from 'int Adapter<Abs, Aspect>::method1()
               [with Abs = Abstraction, Aspect = main()::AspectList]
test.cc:60:    instantiated from here
abstraction.h:12: error: invalid use of member 'Abstraction::_size in static
               member function
forstatic.h:48: error: from this location
forstatic.h:48: error : ' EXEC is not a member of '<declaration error>
```

Figura 5.24: Mensagem indicativa de erro em um Metaprograma.

```
line 1 ""

#ifndef __forward_declarations_for_Object_Locked__
#define __forward_declarations_for_Object_Locked__
class Object_Locked;
namespace AC {
}
#endif

#line 1 "object.ah"

#line 1 ""

#ifndef __ac_h_
#define __ac_h_
namespace AC {
  typedef const char* Type;
  enum JPType { CALL = 0x0004, EXECUTION = 0x0008,
                CONSTRUCTION = 0x0010,
                DESTRUCTION = 0x0020 };
  struct Action {
    void **_args;
    void *_result;
    void *_target;
    void *_that;
    void (*_wrapper)(Action &);
    void *_fptr;
    inline void trigger () { _wrapper (*this); }
  };
  struct AnyResultBuffer {};
  template <typename T> struct ResultBuffer : public AnyResultBuffer
  {
    char _data[sizeof (T)];
    ~ResultBuffer () { ((T*)_data)->T::~T(); }
    operator T& () const { return *(T*)_data; }
  };
  template <class Aspect, int Index>
  struct CFlow {
    static int &instance () {
      static int counter = 0;
      return counter;
    }
    CFlow () { instance ()++; }
    ~CFlow () { instance ()--; }
    static bool active () { return instance () > 0; }
  };

}
inline void * operator new (unsigned int, AC::AnyResultBuffer *p)
{ return p; }
inline void operator delete (void *, AC::AnyResultBuffer *p)
{ } // for VC++

#endif // __ac_h_
```

Figura 5.25: Código produzido pelo *weaver* de ASPECTC++.

Capítulo 6

Conclusões

A premissa inicial deste trabalho era de que linguagens modernas de programação orientada a aspectos provariam ser um mecanismo mais eficiente e de mais fácil utilização do que a técnica de Adaptadores de Cenário. Esta premissa não foi confirmada. O que de fato se observou é que existem casos onde Adaptadores de Cenário demonstram vantagens sobre o uso de ASPECTC++. Os resultados da análise de eficiência na geração de código objeto também demonstram vantagens no uso de Adaptadores de Cenário.

Tanto Adaptadores de Cenário quanto ASPECTC++ não se revelaram suficientes para implementar todos os aspectos considerados nos estudos de caso. Isto sugere que nenhuma destas técnicas é uma solução perfeita para Programação Orientada a Aspectos. Em relação a premissa de que ASPECTC++ pudesse substituir com vantagens Adaptadores de Cenário como mecanismo de aplicação de programas de aspecto, conclui-se que: no desenvolvimento de um novo sistema, onde muitas abstrações são passíveis da aplicação de programas de aspectos, a não necessidade da confecção de adaptadores para as interfaces relevantes das abstrações faz de ASPECTC++ uma alternativa interessante; no caso de um sistema previamente desenvolvido com uso de Adaptadores de Cenário não faz sentido a substituição deste modelo por ASPECTC++. A facilidade de integrar Adaptadores de Cenário a um *framework* como meio de obtenção de configuração de aspectos também deve ser levada em consideração na escolha da técnica a ser utilizada.

Concomitante à realização deste trabalho, diversos pontos negativos em ASPECTC++ foram corrigidos e melhorados por seus desenvolvedores. Por exemplo, foi resolvida a incapacidade de diferenciar funções `const` de funções não `const` nas expressões de ponto de corte, assim como foi implementada a sobrecarga de vários operadores. Mecanismos de Programação Genérica de aspectos também foram inclusos em `AspectC++`. Deste modo, não

se descarta a possibilidade deste trabalho ter-se adiantado na pressuposição de que o tempo transcorrido do surgimento de Programação Orientada a Aspectos até o momento tenha sido suficiente para permitir um desenvolvimento completo de AspectC++.

Já a atividade de desenvolvimento de uma solução experimental para aplicação de AspectC++ em classes parametrizadas e Metaprogramação Estática, mostrou-se satisfatória. O pré-processador de *templates* mostrou-se uma solução viável na separação da resolução de Metaprogramação Estática e classes parametrizadas do resto do processo de compilação. A perspectiva de trabalhos futuros sobre a tecnologia desta ferramenta, tais como conversores e C++ para C, mostra-se promissora.

Por fim, considera-se que o fator mais importante para o sucesso no uso de Programação Orientada a Aspectos pode não ser a ferramenta, mas sim uma adequada separação de conceitos durante o projeto do sistema.

Referências Bibliográficas

[1] Michael Siff and Thomas W. Reps. Program Generalization for Software Reuse: From C to C++ In *Foundations of Software Engineering*, 1996, páginas 135-146.

[2] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *International Symposium on Foundations of Software Engineering*, Vienna, Austria, ACM SIGSOFT, Setembro 2001, pages 88–98.

[3] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[4] H. Ossher and P. Tarr Operation-Level Composition: A case in (Join) Point In *AOP98*, pp.28-31

[5] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. GMD Research Series. GMD - Forschungszentrum Informationstechnik número 17, Sankt Augustin, Agosto 2001.

[6] B. Smith B.Smith's definition of "reflection *report on the OOPSLA'90 Workshop on reflection and Metalevel Architecures in Object-Oriented programming*, Addendum to the OOPSLA'90 Proceedings, 1990.

[7] Coplien, James O. Advanced C++ Programming Styles and Idioms Addison-Wesley, Reading, Massachusetts

[8] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. Scenario Adapters: Efficiently Adapting Components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, U.S.A., Julho 2000.

[9] Xerox and Alan Kay Smaltalk Janeiro 2005.[http://www.smaltalk.org]

[10] Shigeru Chiba OpenC++ Agosto 2004. [http://opencxx.sourceforge.net]

[11] Python Software Foundation Python Janeiro 2005.[http://www.python.org/psf]

[12] Homepage of the TRESE Project, University of Twente Composition Filters Fevereiro 2005. [http://www.trese.cs.utwente.nl]

[13] K. Lieberherr Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns PWS Publishing Company, Boston, MA, 1996

[14] Gregor Kiczales and Yvonne Coady Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code FSE,2001

[15] Palo Alto Research Center AspectJ Janeiro 200. [http://www.http://www.parc.com/research/projects/aspectj]

[16] Free Software Foundation. GNU compiler collection. Janeiro 2004. [http://gcc.gnu.org]

[17] Andreas Gal, Wolfgan Schröder-Preikschat, and Olaf Spinczyk. AspectC++: Language Proposal and Prototype Implementation. In *Proceeding of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, U.S.A., Outubro 2001.

[18] Erich Gamma, Richard Helm, Rakph Johnson, and John Vlissides. Design Patterns: Element of Reusable Object-Oriented Software. *Addison-Wesley*, 1995.

[19] Robert Glück and Jesper Jørgensen. An Automatic Program Generator for Multi-Level Specialization. *Lisp and Symbolic Computation*, Julho 1997, 10(2):113–158.

[20] Beuche, Danilo ; Guerrouat, Abdelaziz ; Papajewski, Holger ; Schröder-Preikschat, Wolfgang ; Spinczyk, Olaf ; Spinczyk, Ute The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *2nd IEEE Intern. Symp. on Object-Oriented Real-Time Distributed Computing* (ISORC'99 St. Malo, France May 1999).St. Malo : - , 1999.

[21] William H. Harrison and Harold Ossher Subject-Oriented Programming (a Critique of Pure Objects). In *Proceeding of the 8th Conference on Object-oriented Programming Systems, Languages and Applciations*, pages 411-428, Washington, U.S.A,, Setembro 1993.

[22] J. Neighbors Software construction using components In technical Reposrt TR-160, Deartment Information and Computer Science, University of California, Irvine, 1980

[23] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, Junho 1997. Springer.

[24] David R. Musser and Alexander A. Stepanov. Generic Programming. In *Proceedings of the First International Joint Conference of ISSAC and AAECC*, Lecture Notes in Computer Science, número 358, pages 13–25, Rome, Italy, Julho 1989. Springer.

[25] Shyam Sunder and David R. Musser. A Metaprogramming Approach to Aspect Oriented Programming in C++. *MPOOL'04 ECOOP* ,2004.

[26] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, Março 1976.

[27] Carlo Pescio. Template Metaprogramming: Make Parameterized Integers Portable with this Novel Technique. *C++ Report*, 1997, 9(7):23–26.

[28] P. J. Plauger. The Standard Template Library. *C/C++ Users Journal*, Dezembro 1995, 13(12):20–24.

[29] Bjarne Stroustrup. C++ Programming Language. *IEEE Software (special issue on Multiparadigm Languages and Environments)*, Janeiro 1986, 3(1):71–72.

[30] Todd L. Veldhuizen. C++ Templates as Partial Evaluation. In *Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*,San Antonio, Janeiro 1999.

[31] Todd L. Veldhuizen and Dennis Gannon Active Libraries: Rethinking the roles of compilers and libraries *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*,Yorktown Heights, New York, 1998.

[32] E.W. Dijkstra A Discipline of programming *Prentica Hall, Englewood Cli* , NJ, 1976.

[33] E.W. Dijkstra Structured Programming *Academic Press*, London, U.K., 1969.

[34] M. Golm and J. Kleinöder Metajava In *STJA'97Conference Proceedings*, Teschnische Universität Ilmenau, Erfurt, Alemanha 1997.

[35] F. Buschmann, R. Meunier, H. Rohnert, P.Sommerland, and M. Stal Pattern-Oriented Software Architecture. A System of Patterns John Wiley & Sons Ltd., Chichester, UK, 1996.

[36] M. Ancona and W. Cazzola. The programming Language Io TR DISI-TR-04-02. Universitá di Genova, Maio 2002.

[37] N. Wirth and M. Reiser. Programming in Oberon - Steps Beyond Pascal and Modula Addison-Wesley, 1992.

[38] Hanspeter Mössenböck and Christoph Steindl The Oberon-2 Reflection Model and Its Applications In *Reflection '99: Proceedings of the Second International Conference on Meta-Level Architectures and Reflection* Springer-Verlag. Londo, UK. 1999

[39] G. Kiczales, J. des Rivières, and D.-G. Bobrow *The Art of the Metaobject Protocol* The MIT press, Cambridge, MA, 1991

[40] Todd L. Veldhuizen. Using C++ Template Metaprograms. *C++ Report*, 7(4):36–43, Maio 1995.

[41] Peter Wegner. Classification in Object-oriented Systems. *ACM SIGPLAN Notices*, 21(10):173–182, Outubro 1986.

[42] David M. Weiss and Chi Tau Robert Lai. *Software Product-line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[43] Michael VanHilst and David Notkin. Using Role Components to Implement Collaboration-Based Designs. *OOPSLA'96 Proceedings*,1996.