Pontifícia Universidade Católica do Rio Grande do Sul

Faculdade de Informática

Programa de Pós-Graduação em Ciência da Computação

Leonardo Penczek

AFR: Uma Abordagem para a Sistematização do Reúso de Frameworks Orientados a Aspectos

Porto Alegre
2006

Leonardo Penczek

AFR: Uma Abordagem para a Sistematização do Reúso de Frameworks Orientados a Aspectos

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre, pelo Programa de Pós-graduação em Ciência da Computação da Faculdade de Informática da Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Toacy Cavalcante de Oliveira

Porto Alegre
2006

Dados Internacionais de Catalogação na Publicação (CIP)

P397a Penczek, Leonardo
AFR: uma abordagem para a sistematização do reúso de frameworks orientados a aspectos / Leonardo Penczek. – Porto Alegre, 2006.
139 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Toacy Cavalcante de Oliveira.

1. Informática. 2. Frameworks Orientados a Aspectos. 3. Reuso de Frameworks. I. Título.

CDD 005.1

Ficha Catalográfica elaborada pelo
Setor de Processamento Técnico da BC-PUCRS

Campus Central
Av. Ipiranga, 6681 – prédio 16 – CEP 90619-900
Porto Alegre – RS – Brasil
Fone: +55 (51) 3320-3544 – Fax: +55 (51) 3320-3548
Email: bceadm@pucrs.br
www.pucrs.br/biblioteca

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada ***"AFR: Uma Abordagem para a Sistematização do Reúso de Frameworks Orientados a Aspectos"***, apresentada por Leonardo Penczek, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Sistemas de Informação, aprovada em 16/03/2007 pela Comissão Examinadora:

Prof. Dr. Toacy Cavalcante de Oliveira – Orientador — PPGCC/PUCRS

Prof. Dr. Marcelo Blois Ribeiro – PPGCC/PUCRS

Prof. Dr. Paulo César Masieiro – USP

Prof. Dr. Ségio Crespo Coelho da Silva Pinto – UNISINOS

Homologada em 07/05/2007, conforme Ata No. 010 pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P. 32 – sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@inf.pucrs.br
www.pucrs.br/facin/pos

AGRADECIMENTOS

Gostaria de agradecer ao meu orientador, professor Toacy, por ter me escolhido como seu orientando. Ele deu a idéia fundamental que resultou na abordagem aqui apresentada bem como todo o apoio necessário durante todo o meu mestrado. Suas idéias, críticas, dúvidas e sugestões influenciaram de forma decisiva o resultado aqui obtido.

Agradeço também à Pontifícia Universidade Católica do Rio Grande do Sul pelo ensino de qualidade e a todos os demais professores do PPGCC pela oportunidade de aprender coisas novas e de ver coisas antigas por um outro ponto de vista.

Meus colegas de mestrado também foram importantes no decorrer do curso, pelos trabalhos em grupo das disciplinas e pelas conversas sobre a vida, e todos eles podem se sentir devidamente agradecidos. Levarei alguns como amigos por toda a vida.

À minha família, meus sinceros agradecimentos, principalmente aos meus pais, Beatriz e Fernando. Foram vocês que sempre acreditaram em mim, que me deram todo o amor possível e todas as condições necessárias para que eu chegasse até aqui.

Agradeço à minha esposa Aline por todo o seu companheirismo, dedicação e paciência, principalmente nos finais de semana “perdidos”, empregados na elaboração deste trabalho.

RESUMO

Frameworks orientados a objetos são muito utilizados atualmente pela sua capacidade de gerar sistemas inteiros de forma muito rápida, por um processo de reúso também chamado de instanciação. Esse processo geralmente não é trivial, sendo necessária a sua documentação para a correta criação de aplicações.

A programação orientada a aspectos introduziu novas possibilidades para o desenvolvimento de frameworks devidos a seus mecanismos de composição. Apesar disso, a introdução de aspectos nos frameworks também tornou o processo de reúso mais complexo, incluindo uma nova etapa de composição além da tradicional etapa de instanciação dos frameworks orientados a objetos. Portanto, se um framework orientado a aspectos não possuir sua estrutura, seus pontos de extensão e seu processo de reúso bem documentados, será muito difícil a sua correta reutilização por parte dos desenvolvedores de aplicação.

Tendo isso em mente, este trabalho tem como objetivo apresentar uma abordagem, denominada AFR (*Aspect-oriented Framework Reuse*), que realiza a sistematização do processo de reúso dos frameworks orientados a aspectos. Para tanto, este trabalho irá propor um conjunto de tecnologias: a notação UML-AFR para a documentação de pontos de extensão, a linguagem RDL+Aspects para descrição das atividades envolvidas no processo de reúso (tanto instanciação quanto composição) e a ferramenta *Reuse Tool* para execução assistida deste processo. Essas tecnologias estão integradas de modo a auxiliar o desenvolvedor de aplicação durante o reúso do framework.

ABSTRACT

Nowadays, object-oriented frameworks are being intensively used in software development as they are able to generate entire systems in a fast way by a reuse process. This process is also called instantiation and usually it is not trivial; it must be well documented to allow the correct creation of applications.

Aspect-oriented programming has introduced new possibilities for the framework development by its composition mechanisms, but it also brought a more complex reuse process, including a new composition phase besides purely instantiation of object-oriented frameworks. Therefore, if an aspect-oriented framework does not have well documented structure, hotspots and reuse steps, it becomes very difficult for the application developer to correctly reuse it.

With this in mind, this work presents an approach, called AFR (Aspect-oriented Framework Reuse) that systemizes the aspect-oriented frameworks reuse. It is composed of a set of technologies: the UML-AFR notation for hotspots documentation, the RDL+Aspects description language for reuse (instantiation and composition) activities and the Reuse Tool that offers a runtime environment for the reuse process. All these technologies are integrated to help the application developer go throw the framework's reuse process.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

ACM	***Association for Computing Machinery***
AFR	***Aspect-oriented Framework Reuse***
AOAsia	***Asian Workshop on Aspect-Oriented Software Development***
AODM	***Aspect-Oriented Design Model***
AOP	***Aspect-Oriented Programming***
CASE	***Computer Aided Systems Engineering***
DAO	***Data Access Object***
EJB	***Enterprise Java Beans***
FAOA	***Framework de Aplicação Orientado a Aspectos***
FOA(s)	***Framework(s) Orientado(s) a Aspectos***
FOO(s)	***Framework(s) Orientado(s) a Objetos***
FRED	***Framework Editor***
FT(s)	***Framework(s) Transversal(is)***
JSF	***Java Server Faces***
JavaEE	***Java Enterprise Edition***
JavaSE	***Java Standard Edition***
LDAP	***Lightweight Directory Access Protocol***
LPG	***Linguagens de Procedimentos Generalizados***
MVC	***Model-View-Controlller***
OBS	***On Board Software Instantiation Environment***
OCL	***Object Constraint Language***
ODBC	***Open DataBase Connectivity***
OO	***Orientado a Objetos***
OOPSLA	***International Conference on Object-Oriented Programming Systems, Languages and Applications***
ORB	***Object Request Broker***
POA	***Programação Orientada a Aspectos***
POO	***Programação Orientada a Objetos***
RDL	***Reuse Description Language***
RDL+Aspects	***Reuse Description Language enhanced with Aspects***
SBES	***Simpósio Brasileiro de Engenharia de Software***

SGBD	***Sistema de Gerenciamento de Banco de Dados***
SIGPLAN	***Special Interest Group on Programming Languages***
SQL	***Structured Query Language***
UML	***Unified Modeling Language***
UML-AFR	***Unified Modeling Language for Aspect-oriented Framework Reuse***
UML-AOF	***Unified Modeling Language for Aspect-Oriented Frameworks***
UML-F	***Unified Modeling Language para Frameworks***
UML-FI	***Unified Modeling Language for Framework Instantiation***
UMLAUT	***UML Transformation Framework***
VCL	***Visual Component Library***
WASP	***Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos***
WTES	***Workshop de Teses e Dissertações***
XMI	***XML Metadata Interchange***
XML	***Extensible Markup Language***
aSideML	***aSide Modeling Language***
xFIT	***Framework Instantiation Tool***

SUMÁRIO

1 INTRODUÇÃO

As próximas subseções contextualizam o trabalho aqui desenvolvido, mostrando a motivação do desenvolvimento de uma abordagem para a sistematização do reúso de frameworks orientados a aspectos. A seguir, enumeram os principais objetivos da abordagem proposta, descrevem a metodologia utilizada e exibem a evolução do trabalho com as publicações geradas no decorrer deste trabalho. Por fim, descrevem a estrutura do texto.

1.1 CONTEXTUALIZAÇÃO

Um dos grandes desafios da Engenharia de Software atualmente é a incorporação de técnicas que aumentem a produtividade e a qualidade no processo de desenvolvimento de software. Um exemplo disso são as técnicas de reúso de conhecimento (projeto e/ou código) adquirido em situações similares, que se destacam por serem eficazes na diminuição de fatores como custo e tempo. O reúso de software possui diversos pontos vantajosos para o desenvolvimento de sistemas, tais como:

- Diminuição do tempo de desenvolvimento, uma vez que o artefato reutilizável representa uma parte já desenvolvida do sistema como um todo.
- Melhoria da qualidade do produto final, uma vez que o artefato reutilizável já passou por uma fase de teste.
- Aumento do vocabulário da linguagem utilizada durante o processo de desenvolvimento de software, uma vez que o artefato reutilizável introduz sua própria linguagem para representar conceitos mais abrangentes (conceitos reutilizáveis).
- Diminuição do custo total de desenvolvimento, dado a diminuição do tempo construção e redução do tempo de manutenção do software.

Essas técnicas de reúso de software vêm evoluindo bastante desde o final dos anos 60, com o advento de procedimentos, funções e bibliotecas, até padrões de projeto [GAM95], programação orientada a aspectos [KIC97a] e frameworks [PRE95][FAY99a][FAY99b] [FAY99c], passando por programação orientada a objetos, componentes, sistemas gerativos [BIG89], entre outras. Destas, o uso de frameworks é uma das mais difundidas atualmente, pois sua utilização permite a geração de sistemas inteiros de forma muito rápida.

Frameworks orientados a objetos (FOOs), ou simplesmente frameworks, são construídos de forma a se reutilizar tanto o código quanto o projeto de uma solução, sendo assim uma espécie de "esqueleto" de aplicações. Frameworks possuem o código genérico comum a todas as aplicações de seu domínio e pontos abstratos que devem ser preenchidos de acordo com os requisitos específicos da aplicação final. Para isso, é necessária a realização de um processo de reúso, que concretizará esses pontos abstratos – também chamados de pontos de extensão – gerando assim a aplicação.

A implementação desses pontos de extensão geralmente utiliza parametrização e/ou características próprias das linguagens orientadas a objetos, como herança, redefinição de métodos, métodos abstratos, etc., além de padrões de projeto voltados para orientação a objetos. O projeto, a construção e a utilização dos pontos de extensão são partes essenciais de todo framework e devem ser muito bem documentados, de forma que o conhecimento possa ser transmitido ao desenvolvedor da aplicação. Embora pareça uma idéia simples, o processo envolvido durante a aplicação do reúso de software encontra diversos obstáculos [JAC97], que exigem uma abordagem para a sistematização deste processo a fim de conseguir de forma eficaz os benefícios associados ao reúso.

Existem diversas abordagens propostas com o objetivo de documentar e/ou sistematizar o reúso de frameworks orientados a objetos, das quais podemos citar *Cookbooks* [KRA88], *Motifs* [LAJ94], *Patterns* [JOH92], *Hooks* [FRO97], *Smartbooks* [ORT00], OBS [CEC03], FRED [HAK01], UMLAUT [HO 99], UML-F [FON99], UML-FI e RDL [OLI01][OLI04] [MEN05].

O desenvolvimento de frameworks só se tornou possível com o advento da programação orientada a objetos. As linguagens orientadas a objetos possibilitam mais reúso de código que as linguagens estruturadas, e também oferecem os mecanismos necessários para a criação de pontos de extensão e a inversão de controle, como herança, polimorfismo, interfaces e classes e métodos abstratos. Apesar disso, a programação orientada a objetos possui certas limitações para abstrair interesses que afetem todo um sistema, como persistência de objetos, controle de acesso, *logging*, desempenho, etc.

A programação orientada a aspectos [KIC97a] surgiu com o objetivo de separar claramente esses interesses do código funcional em estruturas chamadas de aspectos. Esses aspectos devem ser então combinados com o código da aplicação para que a funcionalidade definida nos mesmos possa ser incorporada pelo sistema.

Logo começou-se a investigar o uso desses novos mecanismos de abstração e composição no desenvolvimento de frameworks, dando origem a novas formas de projeto e

construção de pontos de extensão. Isso fornece aos frameworks orientados a aspectos (FOAs), ou seja, frameworks que utilizam aspectos em sua estrutura, características que os diferenciam dos tradicionais frameworks orientados a objetos, como a possibilidade de composição com uma aplicação já existente.

1.2 MOTIVAÇÃO

A introdução de aspectos no desenvolvimento de frameworks criou novas formas de construção de variabilidades, além de aumentar a capacidade de atuação de um framework pela sua composição com algum código já existente. Justamente por essas novas formas de variabilidade e de uso, o processo de reúso dos FOAs se tornou mais complexo, incluindo uma nova etapa de composição além da tradicional etapa de instanciação. Além disso, devido ao fato da orientação a aspectos ser um paradigma complementar ao da orientação a objetos, todas as tradicionais formas de construção de variabilidades orientadas a objetos continuam podendo ser utilizadas no desenvolvimento de um FOA.

Portanto, se um framework orientado a aspectos não possuir sua estrutura, seus pontos de extensão e seu processo de reúso muito bem documentados, será muito difícil a sua correta reutilização por parte dos desenvolvedores de aplicação. Essa documentação deve suportar as variabilidades já existentes orientadas a objetos e as novas introduzidas pela orientação a aspectos.

As abordagens existentes para a documentação de frameworks e de seus processos de reúso são focadas em frameworks orientados a objetos, não levando em consideração as novas características que a orientação a aspectos oferece. Deste modo, essas abordagens conseguem documentar apenas a parte orientada a objetos de um FOA bem como a etapa de instanciação do reúso, mas deixando de fora as variabilidades orientadas a aspectos e a etapa de composição.

1.3 OBJETIVOS

Tendo isso em mente, este trabalho tem como objetivo apresentar a abordagem AFR (*Aspect-oriented Framework Reuse*), capaz de realizar a sistematização do processo de reúso

dos frameworks orientados a aspectos. Essa abordagem compreende um conjunto de tecnologias que sistematizam a definição e o uso desses frameworks:

- Documentação precisa dos pontos de extensão vitais para o reúso desses frameworks dentro de diagramas UML com a UML-AFR, que por sua vez estende a UML-FI [OLI01][OLI04].
- A linguagem RDL+Aspects (*Reuse Description Language enhanced with Aspects*) capaz de capturar formalmente as atividades de reúso relacionadas a esses frameworks, que revisa e estende a RDL (*Reuse Description Language*) [OLI01][OLI04][MEN05].
- A execução assistida do processo de reúso (instanciação e composição) pela ferramenta *Reuse Tool*, que fornece um ambiente de execução para os programas escritos em RDL+Aspects.

1.4 METODOLOGIA

O desenvolvimento da abordagem AFR foi realizado em etapas. Na primeira foi realizado um trabalho de pesquisa na bibliografia disponível a respeito de informações pertinentes a frameworks (FOOs e FOAs) e seu processo de reúso. Isso permitiu a obtenção de diversas abordagens que visam à documentação de frameworks orientados a objetos e seu reúso, mas nenhuma abordagem com foco voltado para frameworks orientados a aspectos. Também foi possível verificar a falta de padronização para a modelagem de aspectos, pois a UML, até a sua versão mais recente, não suporta aspectos: diversas abordagens distintas foram propostas, em sua maioria em conjunto com a UML, mas nenhuma até o presente momento ganhou força suficiente para se tornar um padrão. Aliado a esse fato, as abordagens existentes ainda não conseguiram capturar de uma forma padronizada, clara e eficiente as informações pertinentes ao projeto de pontos de extensão de frameworks orientados a aspectos, como declarações intertipo, herança entre aspectos e conjuntos de junção.

Ainda nessa primeira etapa foi encontrado que o processo de reúso dos FOAs é mais complexo que o dos FOOs, pois, além da instanciação, pode existir uma etapa adicional de composição, na qual é definido como o código do framework (ou da sua instância) será acoplado em um código-base existente. Além disso, FOAs podem utilizar internamente orientação a objetos, e implementar variabilidades da mesma forma que FOOs. Assim sendo,

foi definida como base deste trabalho a abordagem RDL, que possui um conjunto de tecnologias para sistematizar o reúso de FOOs.

Na segunda etapa foram criadas as tecnologias relacionadas ao reúso dos FOAs, tendo como base as tecnologias da RDL. Essas tecnologias foram modificadas e estendidas de acordo com as necessidades para a sistematização do reúso dos FOAs, dando origem à notação UML-AFR, à linguagem RDL+Aspects e ao projeto da ferramenta *Reuse Tool*.

Na terceira etapa, com a implementação da ferramenta, foram realizadas provas de conceito da abordagem para testar o uso da abordagem com alguns FOAs, com a construção de programas de reúso e da execução dos mesmos gerando pequenas aplicações.

A evolução deste trabalho pode ser verificada pelos trabalhos publicados:

- ***Sistematização da instanciação de frameworks orientados a aspectos*** [PEN06a] – artigo aceito pelo *Workshop de Teses e Dissertações* (WTES) que ocorreu em conjunto ao *Simpósio Brasileiro de Engenharia de Software* (SBES). Relata o resultado das pesquisas realizadas até o momento e um primeiro esboço da solução aqui proposta.
- ***Systemizing aspect-oriented framework reuse with AFR*** [PEN06b] – pôster aceito pelo *ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA) mostrando uma visão geral da abordagem AFR, de forma visual, já com suas tecnologias (UML-AFR, RDL+Aspects e *Reuse Tool*) definidas e mapeadas.
- ***AFR: an Approach to Systematize Aspect-Oriented Framework Reuse*** [PEN06c] – artigo aceito pelo *2nd Asian Workshop on Aspect-Oriented Software Development* (AOAsia) descrevendo com detalhes o funcionamento da abordagem como um todo.
- ***RDL+Aspects: uma linguagem de processo para sistematizar o reúso de frameworks orientados a aspectos*** [PEN06d] – artigo aceito pelo *III Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos* (WASP), que ocorreu em conjunto ao SBES, com foco na linguagem RDL+Aspects, descrevendo com mais detalhes os seus comandos relacionados aos elementos da orientação a aspectos.

1.5 ESTRUTURA DO TEXTO

No Capítulo 2 são apresentados os conceitos de reúso de software necessários para o entendimento deste trabalho: frameworks orientados a objetos, programação orientada a aspectos e frameworks orientados a aspectos, juntamente com os trabalhos existentes diretamente relacionados com o processo de reúso de frameworks. O objetivo é mostrar o panorama atual dos assuntos abordados e contextualizar a abordagem proposta neste trabalho com o processo de reúso dos FOAs.

O Capítulo 3 procura definir os requisitos necessários para a documentação de FOAs. Por ser de fundamental importância para a documentação, as técnicas de modelagem orientada a aspectos são explicadas em maiores detalhes, principalmente suas vantagens e desvantagens, mostrando a situação atual deste campo de pesquisa. A seguir, é apresentada a notação UML-AFR, que visa documentar os pontos de extensão de FOAs no nível de projeto.

O Capítulo 4 trata da especificação do processo de reúso de um FOA, descrevendo em detalhes a linguagem RDL+Aspects: sua estrutura e seu conjunto de comandos. Cada comando também é explicado em detalhes: sua sintaxe, sua semântica e a ilustração de sua execução com um exemplo.

A ferramenta *Reuse Tool* é descrita no Capítulo 5: seu funcionamento, sua arquitetura e projeto, além de sua interação com o usuário (desenvolvedor da aplicação – reutilizador).

O Capítulo 6 ilustra o uso da abordagem AFR com alguns frameworks orientados a aspectos, procurando validar a solução proposta em situações práticas. Para cada framework foram gerados os programas de reúso adequados e seus modelos foram anotados com a UML-AFR. Logo após, foram geradas aplicações de demonstração a partir da execução desses programas de reúso.

Por fim, o Capítulo 7 apresentará as conclusões obtidas com o desenvolvimento deste trabalho e as propostas de alguns trabalhos futuros que poderão ser desenvolvidos.

2 REÚSO DE SOFTWARE

O reúso de software vem sendo tratado, desde o final dos anos 60, como uma das principais formas para ser melhorar a qualidade e a produtividade do desenvolvimento de software, pois evita o trabalho recorrente pela reaplicação do conhecimento previamente adquirido no desenvolvimento de novas aplicações [BIG89]. Os principais benefícios são:

- Redução do tempo de desenvolvimento.
- Diminuição da taxa de erros.
- Diminuição do custo total de desenvolvimento.
- Aumento da capacidade de produção de software.
- Formação de especialistas em domínios de aplicação específicos.

O reúso de software começou, historicamente, com a introdução de procedimentos e funções nas linguagens de programação, e, desde então, diversas tecnologias surgiram: bibliotecas, programação orientada a objetos, componentes, sistemas gerativos [BIG89], padrões de projeto [GAM95], frameworks orientados a objetos [PRE95][FAY99a][FAY99b] [FAY99c], programação orientada a aspectos [KIC97a], frameworks orientados a aspectos [CAM04a][CAM05], entre outras. Destas, o uso de frameworks é uma das mais difundidas atualmente, pois permite a geração de sistemas inteiros de forma muito rápida.

Embora pareça uma idéia simples, o processo envolvido durante a aplicação do reúso de software encontra diversos obstáculos [JAC97] que exigem uma abordagem para a sua sistematização. Somente deste modo é possível conseguir, de forma eficaz, os benefícios associados ao reúso de software.

Nas subseções a seguir serão apresentadas as tecnologias de reúso que mais se relacionam com a abordagem proposta neste trabalho: frameworks orientados a objetos, programação orientada a aspectos e frameworks orientados a aspectos.

2.1 FRAMEWORKS ORIENTADOS A OBJETOS

Como mostrado anteriormente, o reúso de software começou com a introdução de bibliotecas de funções e procedimentos, e posteriormente, com o advento da programação orientada a objetos, bibliotecas de classes. O foco principal dessas bibliotecas é a reutilização de código, já que podem ser ligadas em tempo de compilação ou execução à aplicação.

Porém, a reutilização de código é um tanto restritiva, pois durante o processo de implementação de uma determinada abstração, suas idéias originais são normalmente intercaladas e escondidas pelo idioma da linguagem de programação utilizada, não permitindo que todo ou parte do conhecimento adquirido durante o processo de desenvolvimento seja reaproveitado em outras situações. A não ser que o domínio de aplicação seja bem conhecido (p.ex., interfaces gráficas), o alto custo de desenvolvimento de tais bibliotecas e a baixa probabilidade de utilidade desencorajam seu desenvolvimento [JAC97].

Como as idéias (abstração) que estão por trás de um artefato de software também podem ser úteis em diversas situações, uma nova técnica de reúso – padrões de projeto [GAM95] – surgiu com o intuito de capturar essas idéias para serem reutilizadas em situações semelhantes. Gamma [GAM95] definiu que padrões de projeto são descrições de objetos e classes comunicantes que são customizados para resolver um problema geral de projeto em um contexto particular.

Em uma primeira análise, o uso de frameworks é uma abordagem para reutilização tanto de código quanto de projeto, e vem se consolidando como uma das principais técnicas de reúso de software. Frameworks são atraentes porque permitem a geração de sistemas inteiros de forma muito rápida, por um processo de reúso (Figura 1), chamado freqüentemente de processo de instanciação no caso de frameworks orientados a objetos.

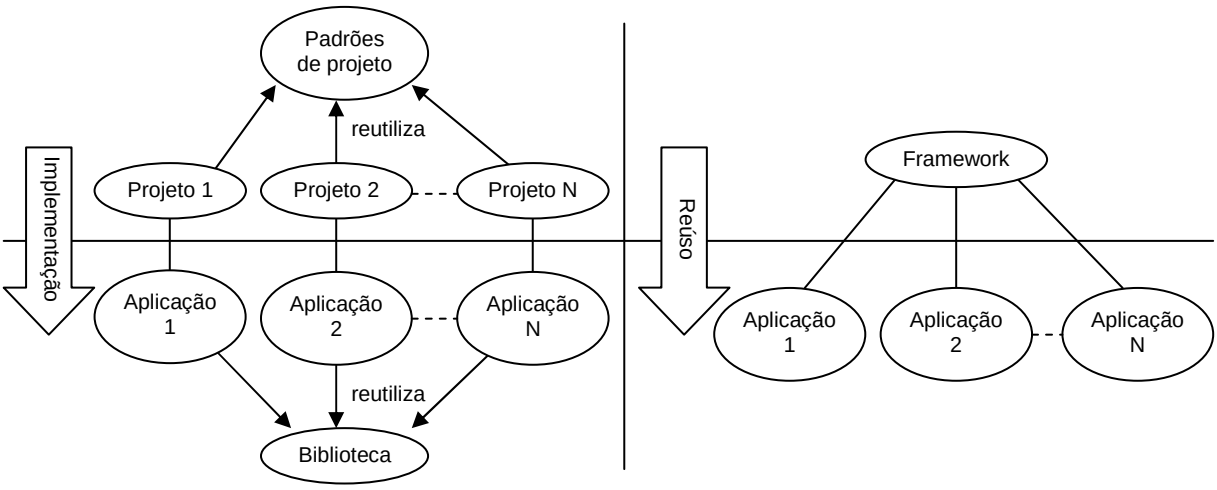


Figura 1 – Bibliotecas, padrões de projeto e frameworks.
Fonte: O autor.

Existem diversas definições de frameworks. Segundo Johnson [JOH88][JOH97], “um framework é um esqueleto de uma aplicação que deve ser parametrizado pelo desenvolvedor da aplicação” e “um framework é um conjunto de classes que representa um projeto abstrato para soluções em uma família de aplicações”. Pode-se notar que a primeira afirmação está

realçando utilidade de um framework, isto é, ser parametrizado durante o processo de reúso. A segunda afirmação trata mais o aspecto estrutural (classes) do framework bem como o direcionamento para um domínio específico.

Pree [PRE95] ressalta que frameworks são "constituídos por pedaços de software semi-acabados e prontos para usar, sendo que reutilizar um framework significa adaptar estes pedaços para uma necessidade específica, pela redefinição de métodos e de algumas classes". A grande contribuição trazida por este trabalho é sem dúvida a definição do termo *hotspot*, que permite identificar de forma clara os pontos de extensão de um framework que servirão de base para o processo de reúso.

Em [FON99] foi afirmado que "um framework é definido como um sistema composto de um subsistema núcleo, que é comum a todas as aplicações instanciadas a partir deste framework, e um subsistema de *hotspots* que implementa o comportamento especifico das aplicações instanciadas. Sendo assim, um desenvolvedor de uma aplicação gera uma instância do framework adequando o subsistema de *hotspots* durante o processo de instanciação". Claramente essa definição está baseada nos conceitos introduzidos em [PRE95] e enfatiza a utilidade de frameworks proposta em [JOH88].

Um dos trabalhos mais completos na área de frameworks foi elaborado por Fayad [FAY99a][FAY99b][FAY99c]. Esse trabalho é composto de uma coletânea de artigos organizados em três livros que relatam a obtenção, desenvolvimento, documentação, evolução e experiências na área de frameworks. Fayad explora o termo frameworks de aplicações no qual "frameworks de aplicação orientados a objetos são uma tecnologia promissora para materializar projetos e implementações de softwares comprovados, levando a redução de custo e ao aumento a qualidade do software". O interessante nessa definição é que ela abrange de forma sucinta os conceitos mais importantes por trás da tecnologia de frameworks: reúso de projeto e de código e orientação a objetos.

É importante ressaltar que o desenvolvimento de frameworks só se tornou possível com o advento da programação orientada a objetos, que trouxe consigo conceitos como herança, interfaces, polimorfismo, abstrações e redefinições, elementos básicos para a construção dos pontos de extensão. Portanto, o termo "framework" isolado refere-se a frameworks orientados a objetos, compostos de classes e interfaces que encapsulam projeto e código a fim de serem reaproveitados em diferentes situações. Com o posterior surgimento da programação orientada a aspectos [KIC97a], surgiram também os frameworks orientados a aspectos [CAM04a][CAM05], que utilizam unidades da programação orientada a aspectos em sua estrutura, e serão tratados mais adiante neste capítulo, na Subseção 2.3.

2.1.1 Vantagens e Desvantagens

Segundo [FAY99a], os principais benefícios da utilização de frameworks são:

- **Modularidade** – o sistema resultante possui uma alta modularidade, devido ao encapsulamento dos detalhes de implementação por trás de interfaces estáveis. Essa modularidade torna possível incrementar a qualidade do software, uma vez que os impactos causados por alterações de projeto e implementação são localizados, reduzindo o esforço necessário para o entendimento e manutenção do software existente.
- **Reusabilidade** – o uso de interfaces estáveis incentiva a reusabilidade uma vez que definem um comportamento conhecido, que pode ser reaplicado para criar novas aplicações. Essa reusabilidade carrega o conhecimento em um domínio e o esforço anterior de desenvolvedores experientes, que já foi testado e comprovado, evitando a criação de uma nova solução para um problema recorrente.
- **Extensibilidade** – com a definição de pontos de extensão que permitem a uma aplicação estender suas interfaces estáveis. Esses pontos de extensão permitem o desacoplamento sistemático da parte fixa do framework, presente no domínio da aplicação, da parte variável introduzida pelo processo de instanciação.
- **Inversão de controle** – uma novidade trazida pela reutilização de frameworks é a possibilidade da inversão do fluxo de controle, isto é, quem comanda o fluxo de execução principal do programa é o artefato reutilizável e não o artefato reutilizador (Figura 2). Esse conceito permite que um framework especifique seu funcionamento como um todo, principalmente no que se refere à coordenação de seus componentes principais. Em abordagens convencionais como as encontradas na reutilização de bibliotecas e componentes o artefato reutilizável é passivo, o que torna seu desenvolvimento muito mais complexo uma vez que o contexto em que este artefato será inserido é totalmente desconhecido pelo projetista. A inversão de controle está intimamente ligada aos mecanismos de extensão presentes em linguagens orientadas a objetos nos quais frameworks se baseiam. Esses mecanismos, como polimorfismo e *late-binding*, permitem que objetos "executem" um código a ser definido futuramente pelo desenvolvedor da aplicação, durante o processo de reúso. Essa execução se dá por um protocolo bem definido, normalmente especificado pelo

mecanismo de herança. É essa inversão de controle que, em última análise, possibilita a criação dos esqueletos de aplicação mencionados por Johnson [JOH88].

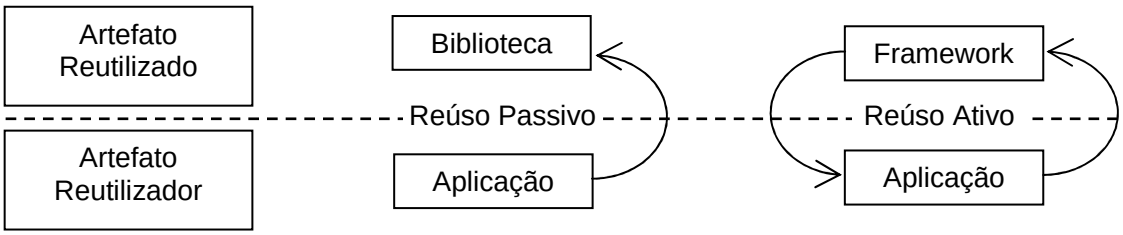


Figura 2 – Inversão de controle pelo uso de frameworks.
Fonte: Oliveira [OLI01].

Embora o uso de frameworks seja bastante vantajoso, sua aplicação na prática possui alguns inconvenientes:

- **Esforço de desenvolvimento** – como o desenvolvimento de sistemas complexos é difícil, o desenvolvimento de sistemas de forma abstrata tendo em mente sua reutilização é mais difícil ainda. São necessários tempo, recursos, excelente domínio de reúso de software e de orientação a objetos, experiência no domínio para o qual o framework está sendo desenvolvido, bem como uma boa dose de criatividade.
- **Curva de aprendizado** – um dos maiores problemas é o tempo necessário para se aprender o funcionamento e o processo de reúso, a fim de se obter as vantagens da utilização de determinado framework. Sendo assim, se o custo dessa aprendizagem não for amortizado por vários projetos ou se o ganho de produtividade e qualidade não for expressivo, o investimento em frameworks não se tornará atraente.
- **Integrabilidade** – A maioria dos frameworks é desenvolvida exclusivamente para o propósito de extensão, e não com o de integração com outros artefatos de software. Sendo assim, problemas difíceis de serem solucionados, como, por exemplo, quem comanda do fluxo de controle da aplicação final, podem surgir e dificultar o processo de integração. Isso é muito comum quando se tenta integrar frameworks [MAT00].
- **Manutenibilidade** – Como todo artefato de software, seus requisitos iniciais evoluem no tempo, obrigando a criação de novas versões do framework. Sendo assim, as aplicações instanciadas a partir de um dado framework também devem evoluir com a finalidade de se manterem de acordo com a especificação do framework. Esse problema está relacionado com a modificação de aplicações que já estão em produção, o que pode ser problemático quando o serviço prestado por estas aplicações não puder parar.

- **Eficiência** – o uso de frameworks usualmente provoca queda na eficiência do código final, uma vez que chamadas adicionais a tabelas virtuais de métodos serão necessárias para executar uma determinada tarefa.

2.1.2 Classificação

Frameworks podem ser classificados de acordo com seu escopo ou forma de extensão. Quanto ao escopo, é observada a camada na qual o framework está atuando, podendo ser [FAY99a]:

- **Infra-estrutura** – simplificam o desenvolvimento de sistemas portáteis e eficientes como sistemas operacionais, comunicações e interface com o usuário. São normalmente utilizados internamente em uma organização e não são vendidos a clientes.
- **Integração** – são comumente utilizados para integrar sistemas distribuídos permitindo a troca de dados entre sistemas heterogêneos. Sistemas compatíveis com o modelo ORB (*Object Request Broker*) são exemplos deste tipo de framework.
- **Domínio específico** – estes frameworks são direcionados para amplos domínios de aplicação como telecomunicações, manufatura e finanças. Em geral são sistemas complexos que envolvem infra-estrutura e integração, tornando-os dispendiosos para serem desenvolvidos.

Quanto à forma de extensão, é observada a técnica utilizada para estender um dado framework, podendo ser:

- **Caixa branca (*Whitebox*)** – são fortemente baseados nas características de linguagens de programação orientadas a objetos como herança, redefinições e *late-binding*, para expressarem/implementarem pontos de extensão. O reutilizador precisa ter profundo conhecimento de sua estrutura, bem como das colaborações dos objetos envolvidos. São comumente utilizados em conjunto com bibliotecas de componentes para facilitar o processo de reúso.
- **Caixa preta (*Blackbox*)** – baseiam seu mecanismo de extensão na composição de objetos. Isto se dá com a definição de interfaces que serão utilizadas para permitir o acoplamento de componentes externos. São mais fáceis de usar que os frameworks

caixa branca, uma vez que o reutilizador não precisa conhecer as entranhas do artefato reutilizável. Freqüentemente são chamados de componentes [MAT00].

- **Caixa cinza (*Graybox*)** – são projetados para evitar as desvantagens presentes em frameworks caixa branca e caixa preta, permitindo certo grau de extensibilidade sem a necessidade de se expor informações internas. Frameworks visuais como o Borland VCL são exemplos, pois permitem o reúso a partir da ligação de componentes e ainda provém parametrização via herança.

2.1.3 Implementação de variabilidades

Os mecanismos de implementação dos pontos de extensão de frameworks orientados a objetos dependem da classificação do mesmo. Frameworks caixa branca utilizam basicamente as características que a programação orientada a objetos fornece [OLI01]:

- **Herança** – uma classe pode ter sua funcionalidade estendida com a introdução de novos métodos em suas classes filhas.
- **Redefinição de métodos** – um método pode ter seu comportamento alterado na classe filha. Muito utilizado em conjunto com métodos abstratos; estes podem ser invocados sem a necessidade de conhecer seu funcionamento, que só será definido na classe filha.
- **Interfaces e classes abstratas** – definem um comportamento que deverá ser implementado no processo de instanciação.
- **Padrões de projeto** – determina a especialização de uma classe ou método pelo uso de padrões de projeto.
- **Seleção de classes** – varia-se o funcionamento de uma classe pela seleção de uma de suas subclasses concretas.
- **Atribuição de valores** – altera-se o funcionamento pela atribuição de valores a determinados atributos de uma classe.

Já frameworks caixa preta são instanciados com parametrização e acoplamento de componentes, geralmente utilizando-se de arquivos de configuração e de chamadas de determinados métodos passando-se os parâmetros desejados. Uma outra maneira, que alguns frameworks fornecem, é a utilização de ajudantes gráficos (*wizards*), programas que vão questionando o usuário sobre os parâmetros necessários para a instanciação.

2.1.4 Processo de Reúso (Instanciação)

O processo de reúso de um framework orientado a objetos é também chamado de processo de instanciação, pois a partir de projeto e código abstratos (framework) é produzida uma instância concreta do framework (aplicação). Durante esse processo os pontos de extensão, que são projetados para serem genéricos e necessitam ser adaptados de acordo com os requisitos de cada aplicação, são preenchidos (Figura 3). Esse processo está fortemente baseado na classificação do framework e pode requerer um maior ou menor conhecimento do artefato reutilizável.

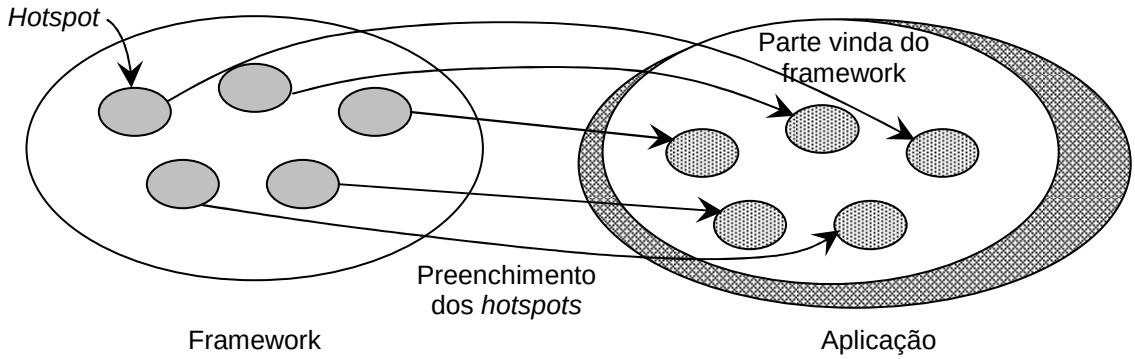


Figura 3 – Instanciação de frameworks orientados a objetos.
Fonte: O autor.

O que pode ser observado durante a instanciação é que o processo tradicional de desenvolvimento de uma aplicação [SOM04] deve ser alterado de modo a incorporar nesta aplicação as características impostas pelo framework. Esse processo começa de forma similar ao desenvolvimento de aplicações comuns com uma fase de requisitos [SOM04], no qual os requisitos funcionais e não funcionais são coletados e expressos em uma notação específica. Em seguida o domínio da aplicação é investigado para se produzir um modelo conceitual do problema em questão. Normalmente, é durante esta fase que frameworks são apresentados como soluções para o problema. Uma vez escolhido o framework, inicia-se o processo de instanciação que tem como objetivo integrar/adaptar o modelo conceitual da aplicação com o modelo de classes presente no framework, no caso de frameworks caixa branca. Em seguida ocorrem os passos tradicionais de codificação e teste.

2.1.4.1 Documentação

Essa integração/adaptação presente na instanciação está fortemente baseada na documentação do framework e é normalmente executada de forma intuitiva. O reutilizador tem que conhecer os ponto de extensão e seu funcionamento para que estes possam ser devidamente estendidos. Diversas abordagens foram propostas com o objetivo de facilitar a compreensão dos pontos de extensão e de suas formas de uso: *Cookbooks* [KRA88], *Motifs* [LAJ94], *Patterns* [JOH92], *Hooks* [FRO97], *Smartbooks* [ORT00], OBS [CEC03], FRED [HAK01], UMLAUT [HO 99], UML-F [FON99], UML-FI e RDL [OLI01][OLI04][MEN05], entre outras. Essas abordagens permitem ao desenvolvedor do framework especificar como o mesmo deve ser instanciado, auxiliando assim o reutilizador durante o processo de reúso.

A abordagem de *Cookbooks* [KRA88] foi proposta como um tutorial para a utilização do framework MVC (*model-view-controlller*) presente na biblioteca de classes de *Smalltalk*. Ele descreve o framework de forma geral, com linguagem natural, usando a idéia de livro de receitas (*cookbook*), e em seguida descreve as partes relevantes à instanciação. Por último apresenta uma série de exemplos que utilizam o framework.

Motifs [LAJ94] e *Patterns* [JOH92] propõem que a experiência e o conhecimento dos desenvolvedores de como o framework deve ser utilizado pode ser capturado com uma série de padrões sendo que cada padrão deve ter nome, problema, solução, exemplo, resumo da solução e padrões relacionados.

Hooks [FRO97] são considerados um aprimoramento de *Cookbooks*, uma vez que descrevem pontos de extensão de forma mais estruturada, detalhando aspectos relevantes como participantes no *hook* e alterações necessárias para seu uso. Em sua estrutura, *hooks* descrevem nome, requisito, tipo, área, usa, participantes, mudanças, restrições e comentários. Embora seja uma forma muito mais precisa de se documentar um framework, *hooks* carecem de informações necessárias para se automatizar o processo de instanciação. Para tal seria necessário obter uma descrição manipulável de ambos, projeto e *hooks*, com a finalidade de ajudar o reutilizador a especificar os refinamentos/parametrizações necessários, além da criação de uma ferramenta que utilizasse as informações dos *hooks* para manipular o projeto.

Smartbooks [ORT00] fazem uso de agentes de software para executarem planos de instanciação. A maior contribuição dessa abordagem é a introdução de uma notação não-padrão – TOON [ORT00] – mas que por sua vez causa uma carga extra ao desenvolvedor do framework.

Em OBS [CEC03] os autores utilizam uma abordagem generativa para a instanciação de frameworks. Porém, essa abordagem é baseada em frameworks caixa preta prontos para uso, em que o processo de instanciação se baseia na configuração de componentes, não sendo personalizável.

FRED [HAK01] é um editor de frameworks que utiliza padrões especializados para gerar aplicações. Contudo, FRED possui seu foco voltado ao código e é fortemente amarrado à linguagem de programação Java [SUN95].

UMLAUT [HO 99] apresenta um framework genérico de transformação UML [OMG06a] baseado em composições algébricas e transformações elementares, que pode ser utilizado para auxiliar a instanciação, mas, por não ser o foco da ferramenta, o reúso acaba se tornando mais complexo do que em outras abordagens.

Em [FON99] foi proposta a UML-F, uma linguagem que estende a UML para capturar os pontos de extensão no nível de projeto e representar o processo de instanciação com os diagramas comportamentais da UML. Essa linguagem permite expressar pontos de extensão como métodos de variação, classes de extensão e interfaces, além de mapear os pontos de extensão existentes em possíveis formas de instanciação, o que permite uma rápida compreensão pelo desenvolvedor da aplicação. Um problema dessa abordagem é relativo à especificação de um diagrama que possa representar completamente o fluxo de atividades de reúso como iterações, condicionais, entre outras.

A UML-FI e a RDL propostas em [OLI01][OLI04][MEN05] fazem parte de uma abordagem que permite a sistematização do processo de instanciação de framework orientados a objetos. Por ser a base deste trabalho, essa abordagem será vista em maiores detalhes na próxima subseção.

2.1.4.2 RDL

A UML-FI [OLI01][OLI04], que estende a UML-F, possui um enfoque maior na instanciação propriamente dita do que na documentação dos pontos de extensão. A UML-FI introduz o conceito de elemento reutilizável, o suporte a regras para parametrização de atributos, a definição de *hotspots* como opcionais e obrigatórios e a especificação dos pontos de extensão de acordo com o tipo de extensão associada.

A RDL (*Reuse Description Language*) [OLI01][OLI04][MEN05] é uma linguagem que descreve de maneira formal o processo de instanciação de um framework orientado a objetos,

permitindo aos desenvolvedores de frameworks representar as tarefas de instanciação explicitamente. A RDL é independente da linguagem de programação e do domínio do framework, manipulando os elementos expressos em UML no nível de projeto. As construções de mais alto nível da RDL são representadas por *cookbooks*, *recipes* e *patterns*. Um *cookbook* contém um conjunto de *recipes*. Uma *recipe* engloba tarefas de instanciação relacionadas a um determinado aspecto variável da arquitetura do framework. Um *pattern* descreve passos recorrentes de instanciação encontrados durante a adaptação do framework, como padrões de projetos. A RDL possui também um conjunto de comandos (Quadro 1) que capturam de maneira formal as principais atividades relacionadas à instanciação de frameworks caixa branca orientado a objetos, sendo possível a construção de programas (Quadro 2a) e de bibliotecas de padrões (Quadro 2b) que podem ser processados por um computador.

Comando	**Descrição**	**Comando**	**Descrição**
`NEW_CLASS(...)`	criação de classe	`variável = expressão`	atribuição
`NEW_METHOD(...)`	criação de método	`CLASS_EXTENSION(...)`	extensão de classe
`NEW_ATTRIBUTE(...)`	criação de atributo	`METHOD_EXTENSION(...)`	extensão de método
`NEW_INHERITANCE(...)`	herança	`VALUE_ASSIGNMENT(...)`	atribuição de valor
`LOOP (e) ... END_LOOP`	comando de repetição	`VALUE_SELECTION(...)`	seleção de valor
`IF (e) ... [ELSE ...] END_IF`	comando condicional	`CALL_PATTERN(..)`	chamada de padrão

Quadro 1 – Comandos principais da RDL.
Fonte: Mendonça [MEN05].

```
COOKBOOK myCookBook;
  RECIPE main;
    ...
    CALL_RECIPE( R1, (...) );
    ...
  END_RECIPE;
  RECIPE R1(...);
    ...
  END_RECIPE;
  ...
END_COOKBOOK;
```
a

```
PATTERN_LIBRARY myPatternLibrary;
  PATTERN Pattern1(...);
    ...
  END_PATTERN;
  PATTERN Pattern2(...);
    ...
  END_PATTERN;
  ...
END_PATTERN_LIBRARY;
```
b

Quadro 2 – Estrutura básica de um programa e de uma biblioteca de padrões RDL.
Fonte: Mendonça [MEN05].

A ferramenta xFIT (*Framework Instantiation Tool*) [OLI01][OLI04][MEN05] dá o suporte computacional à abordagem RDL, oferecendo um ambiente de execução para programas RDL e permitindo a sistematização do processo de instanciação. Além disso, a ferramenta executa tarefas de validação dos elementos de projeto, como, por exemplo, garantir que todas as classes e métodos abstratos tenham sido concretizados no projeto final.

Ao final da execução de um programa de instanciação, a ferramenta gera o modelo correspondente à aplicação final, como pode ser observado na Figura 4.

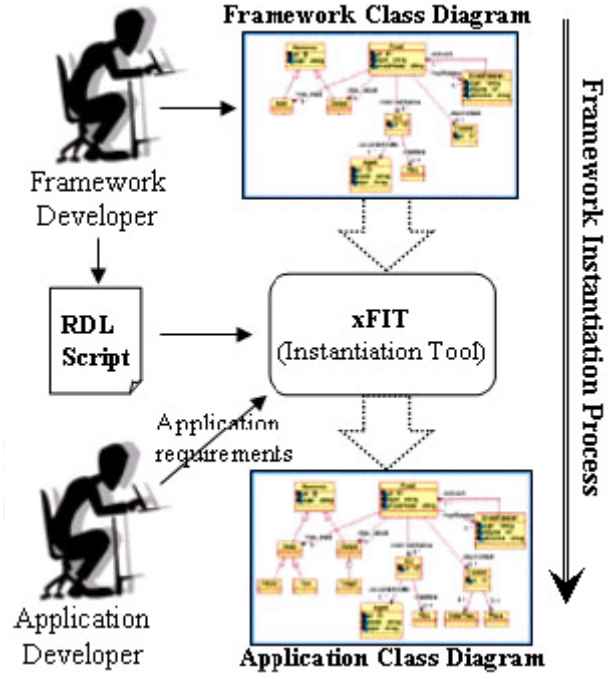


Figura 4 – Visão geral da abordagem RDL.
Fonte: Mendonça [MEN05].

Os passos necessários para a instanciação de um framework orientado a objetos usando a abordagem RDL são:

- O desenvolvedor do framework fornece o diagrama de classes UML do framework no formato XMI (*XML Metadata Interchange*) [OMG06b], com as anotações propostas na UML-FI, e o programa RDL que contém os passos de instanciação do framework.
- O desenvolvedor da aplicação executa a xFIT entrando com o programa RDL e o diagrama de classes do framework. Quando necessário, o desenvolvedor da aplicação alimenta o processo de instanciação, fornecendo informações específicas de acordo com os requisitos da aplicação sendo gerada.
- Ao final do processo de geração, a xFIT executa tarefas de validação e reporta os erros encontrados, caso existam; caso contrário, é produzido um diagrama de classes contendo as classes do framework e as classes específicas da instância gerada.
- O desenvolvedor da aplicação pode utilizar uma ferramenta CASE para abrir o modelo da aplicação e gerar código para as classes produzidas.

2.2 PROGRAMAÇÃO ORIENTADA A ASPECTOS

A programação orientada a objetos (POO) é atualmente o paradigma de programação dominante, por sua capacidade de construir sistemas a partir da decomposição de um problema em objetos e então codificá-los. Esses objetos abstraem dados e comportamento em uma única entidade, e a cooperação entre diversos objetos é responsável pelo funcionamento de todo o sistema. Com isso, a construção de sistemas com grande complexidade com a utilização de um código fonte compreensível e manutenível tem sido possível.

A POO, contudo, possui certas limitações. Existem alguns problemas de programação que nem a orientação a objetos nem a programação procedural são claras em capturar e resolver. Isso porque, segundo Kiczales [KIC97a], linguagens orientadas a objetos, procedurais ou funcionais possuem uma raiz comum: seus mecanismos de abstração e composição são, de uma forma ou de outra, derivações de procedimentos. As metodologias de projeto para essas linguagens tendem a quebrar um sistema em unidades de comportamento ou função (decomposição funcional). A natureza da decomposição varia de acordo com o paradigma, mas cada unidade é encapsulada em um procedimento/função/objeto, e o que foi encapsulado é denominado de *unidade funcional*.

Na prática, um sistema possui tanto requisitos *funcionais*, que fazem parte da lógica de negócio do problema (p.ex, um cadastro de clientes), quanto requisitos *não-funcionais*, que não fazem parte do domínio do problema mas são necessários para o seu funcionamento (persistência, desempenho, segurança, *logging*). Esses requisitos não-funcionais afetam diversas partes do sistema, e são codificados junto com o código responsável por algum requisito funcional. Isso acarreta em alguns problemas [LAD02]:

- **Entrelaçamento de código[1] (*code tangling*)** – um módulo do sistema acaba tendo que implementar, além do requisito funcional, os requisitos não-funcionais envolvidos, como desempenho, sincronização, *logging* e segurança. Essa multiplicidade de requisitos resulta na presença simultânea de diversos interesses[1] (*concerns*) dentro do mesmo código, resultando no entrelaçamento de código.
- **Espalhamento de código[1] (*code scattering*)** – como esses requisitos não-funcionais acabam sendo necessários em todo o sistema, eles também são chamados de interesses transversais[1] (*crosscutting concerns*), e acabam sendo implementados diversas vezes, em diferentes partes do sistema, ocasionando redundância.

[1] Termos de orientação a aspectos seguindo a tradução definida no WASP'04

O Quadro 3 nos mostra um exemplo típico de codificação de um sistema orientado a objetos, no qual o requisito não-funcional de *logging* (linhas 3, 5, 10, 12, 17, 19) fica entrelaçado ao código de negócio e espalhado por todo o sistema, como mostrado graficamente na Figura 5.

```
 1 public class Foo {
 2     public void doBusiness1() {
 3         logger.log("Starting Foo.doBusiness1()");
 4         ... faz alguma regra de negócio ...
 5         logger.log("Ending Foo.doBusiness1()");
 6     }
 7 }
 8 public class Bar {
 9     public void doBusiness2(String p1) {
10         logger.log("Starting Bar.doBusiness2(" + p1 + ")");
11         ... faz alguma regra de negócio ...
12         logger.log("Ending Bar.doBusiness2(" + p1 + ")");
13     }
14 }
15 public class Foobar {
16     public void doBusiness3(Integer p1, String p2) {
17         logger.log("Starting Foobar.doBusiness3(" +p1+ "," +p2+ ")");
18         ... faz alguma regra de negócio ...
19         logger.log("Ending Foobar.doBusiness3(" +p1+ "," +p2+ ")");
20     }
21 }
```

Quadro 3 – Exemplo de classes entrelaçando código de negócios e *logging*.
Fonte: O autor.

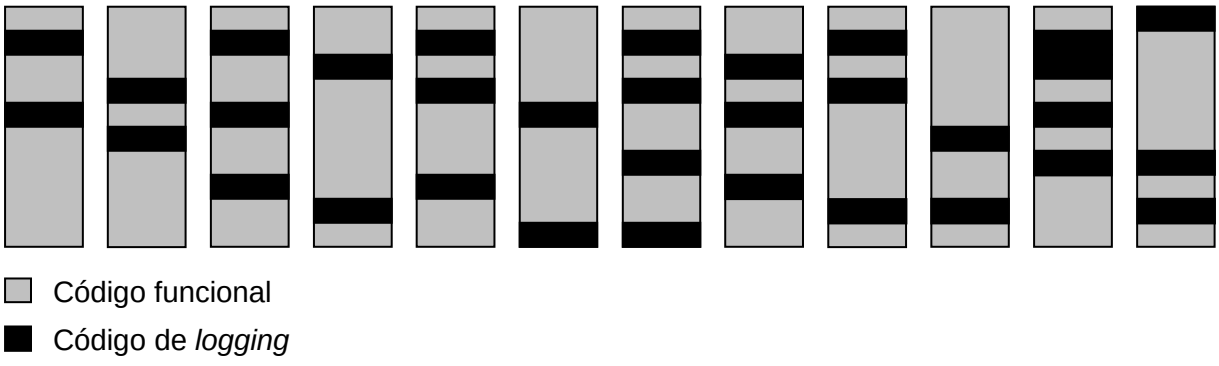


Figura 5 – Representação gráfica do código de *logging* espalhado e entrelaçado.
Fonte: O autor.

O entrelaçamento e o espalhamento de código nos trazem os seguintes problemas:

- **Baixa rastreabilidade** – a codificação simultânea de diversos interesses obscurece a correspondência entre o interesse e a sua implementação, resultando num fraco mapeamento entre os dois.
- **Baixa produtividade** – a codificação simultânea de diversos interesses faz com que o desenvolvedor mude constantemente o foco de seu interesse principal para os interesses secundários.

- **Menos reúso de código** – devido a um módulo implementar diversos interesses, outros sistemas requisitando uma funcionalidade similar não poderão prontamente utilizar o módulo, bem como os interesses não-funcionais, por estarem entrelaçados no código de negócio de um sistema específico. Isso impede o reaproveitamento de módulos em diversos sistemas.
- **Baixa qualidade do código** – o entrelaçamento pode produzir código com problemas "escondidos". Além disso, ao focar em diversos interesses, alguns destes podem não receber a devida importância frente aos demais.
- **Dificuldade de evolução** – uma visão limitada e recursos reduzidos geralmente produzem um projeto que se dirige somente aos interesses atuais. Novos requisitos usualmente necessitam em retrabalho e recodificação. Se esses requisitos forem transversais, isto significa em mexer em diversos módulos, que, por sua vez, pode gerar inconsistências. Para que isso não ocorra, é necessário um esforço considerável em testes a fim de garantir que as mudanças não causem erros.

A resolução desses problemas serviu de motivação para o surgimento de um novo paradigma de programação. A programação orientada a aspectos (POA) [KIC97a][ELR01] foi criada com o propósito de evitar os problemas do entrelaçamento e do espalhamento de código, bem como identificar os interesses transversais e implementá-los de uma forma independente do código funcional. A POA tem suas origens principalmente em protocolos de metaobjetos [KIC96], em implementações abertas [KIC97b], na programação reflexiva [MDK93] e na programação adaptativa [LIE94].

Segundo Kiczales [KIC97a], a principal deficiência das linguagens orientadas a objetos, procedurais e funcionais, as quais são chamadas por Kiczales de linguagens de procedimentos generalizados (LPG), é o seu mecanismo de composição único: chamadas de procedimentos. Com esse mecanismo de composição é possível construir unidades funcionais que possuam apenas um único fluxo, impossibilitando a composição de requisitos funcionais e não-funcionais, pois ambos não seguem um mesmo fluxo, necessitando diferentes regras de composição. Mas como um sistema necessita mesmo assim compor ambos os interesses para funcionar, essa composição é feita manualmente na mesma estrutura pelo programador, gerando um código entrelaçado.

Em geral, quando duas propriedades sendo programadas necessitam ser compostas diferentemente e mesmo assim precisam estar coordenadas, é dito que essas propriedades

entrecortam[2] (*crosscut*) uma a outra. Como as linguagens de composição procedural fornecem um único mecanismo de composição, o programador necessita fazer essa co-composição manualmente, gerando um código complexo e entrelaçado. Isso dá origem a dois termos importantes:

- uma propriedade que necessita ser implementada e pode ser claramente encapsulada em um procedimento generalizado é um **componente**.
- uma propriedade que necessita ser implementada mas não pode ser claramente encapsulada em um procedimento generalizado é um **aspecto**.

A POA tem como objetivo ajudar o programador a separar claramente componentes e aspectos uns dos outros (componentes de aspectos, componentes de componentes e aspectos de aspectos), fornecendo mecanismos que tornam possível a abstração e a composição dos mesmos para produzir o sistema como um todo. A estrutura de implementação baseada em POA de um sistema é análoga à estrutura de implementação baseada em LPG. A implementação de uma aplicação em LPG consiste em:

- uma linguagem de programação
- um compilador ou interpretador para a linguagem
- um programa escrito na linguagem que implementa a aplicação

A aplicação é o resultado gerado pelo compilador/interpretador. Já a implementação baseada em POA de uma aplicação consiste em:

- uma linguagem de componente para programar os componentes
- uma ou mais linguagens de aspectos para programar os aspectos
- um combinador de aspectos[2] (*aspect weaver*) para fazer a composição das linguagens
- um programa de componente, que implementa os componentes utilizando a linguagem de componentes
- um ou mais programas de aspectos, que implementam os aspectos utilizando as linguagens de aspectos

A aplicação, nesse caso, é o resultado da combinação do código dos aspectos com o código dos componentes. Essa combinação dos programas de aspectos com o programa de componentes pode ser tanto feita em tempo de execução (*runtime*) quanto em tempo de compilação, dependendo do tipo de combinador que se utilize. Ambas as estruturas acima são mostradas na Figura 6.

[2] Termos de orientação a aspectos seguindo a tradução definida no WASP'04

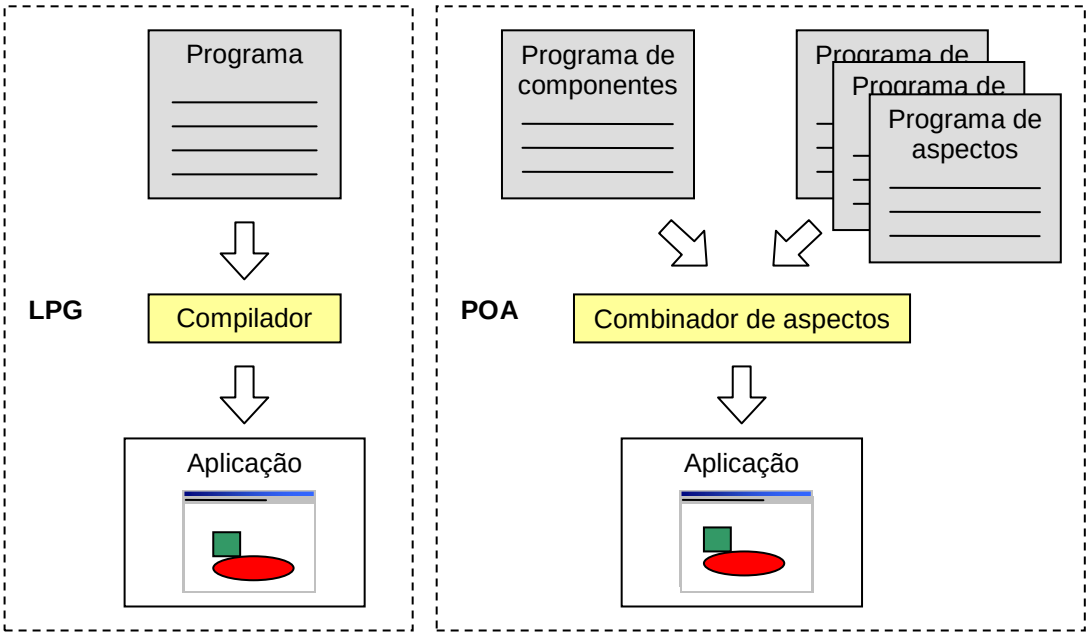


Figura 6 – Implementação LPG vs. implementação POA.
Fonte: O autor.

O projeto de um sistema orientado a aspectos envolve a compreensão do que deve ir para a linguagem de componente, do que deve ir para a linguagem de aspectos e do que deve ser compartilhado entre as linguagens. A linguagem de componente deve permitir ao programador implementar as funcionalidades do sistema, ao mesmo tempo assegurando que estes programas não façam nada do que os programas de aspectos devem controlar. A linguagem de aspectos deve permitir a implementação dos aspectos de uma forma natural e concisa. Ambas as linguagens terão diferentes mecanismos de abstração e composição, mas também deverão ter alguns termos em comum para tornar possível a composição dos programas. Esses termos comuns vão depender da linguagem de componente, bem como dos domínios de aplicação dos componentes e dos aspectos. Para linguagens orientadas a objetos, uma das maneiras possíveis para fazer essa junção entre componentes e aspectos é o acesso reflexivo da invocação de métodos, como mostrado na programação reflexiva [MDK93].

O combinador de aspectos precisa processar tanto a linguagem de componentes quanto a de aspectos, compondo ambas propriamente para produzir a operação desejada do sistema como um todo. Para que o combinador possa fazer isso é fundamental o conceito de pontos de junção[3] (*join points*), que são os elementos da linguagem de componentes que se coordenam com os programas de aspectos. O combinador trabalha gerando uma representação dos pontos

[3] Termos de orientação a aspectos seguindo a tradução definida no WASP'04

de junção da linguagem de componente, e então executando ou compilando os programas de aspectos que estão relacionados ao ponto de junção correspondente.

```
 1 public class Foo {
 2     public void doBusiness1() {
 3         ... faz alguma regra de negócio ...
 4     }
 5 }
 6 public class Bar {
 7     public void doBusiness2(String p1) {
 8         ... faz alguma regra de negócio ...
 9     }
10 }
11 public class Foobar {
12     public void doBusiness3(Integer p1, String p2) {
13         ... faz alguma regra de negócio ...
14     }
15 }
16 aspect Logging {
17     before(): execution(public * *(..)) {
18         logger.log("Starting " + thisJoinPoint.getSignature());
19         Object[] args = thisJoinPoint.getArgs();
20         for (int i = 0; i < os.length; i++) {
21             logger.log("Arg #" + i + " " + args[i]);
22         }
23     }
24     after(): execution(public * *(..)) {
25         logger.log("Ending " + thisJoinPoint.getSignature());
26         Object[] args = thisJoinPoint.getArgs();
27         for (int i = 0; i < os.length; i++) {
28             logger.log("Arg #" + i + " " + args[i]);
29         }
30     }
31 }
```

Quadro 4 – Classes de negócio sem entrelaçamento e aspecto de *logging*.
Fonte: O autor.

Para ilustrar o funcionamento de aspectos, o Quadro 4 mostra o sistema exemplo exibido anteriormente reimplementado utilizando-se um aspecto para realizar o *logging*. Para isso, foi utilizada a linguagem AspectJ [KIC01][XER06], desenvolvida pelo grupo de POA do *Xerox Palo Alto Research Center*, o mesmo grupo a abordar aspectos pela primeira vez [KIC97a]. A AspectJ é uma linguagem de aspectos que complementa a linguagem orientada a objetos Java, e introduz os conceitos de: **conjuntos de junção**[4], ou conjuntos de pontos de junção[4] (*pointcuts*), que identificam pontos de junção pela filtragem de um subconjunto de todos os pontos de junção contidos no curso do programa funcional; **declarações intertipo**[4] (*inter-type declarations*) ou introduções, que definem mudanças (novos métodos, atributos e/ou herança) a serem introduzidas em classes e interfaces; e **adendos**[4] (*advices*), que definem

[4] Termos de orientação a aspectos seguindo a tradução definida no WASP'04

o código adicional a ser executado nos pontos de junção no momento anterior, posterior, ou em substituição à execução destes.

Como pode ser observado, o código de negócio agora fica mais conciso (linhas 1–15), por não precisar do código de *logging* entrelaçado; este não se encontra mais espalhado por todo o sistema mas sim concentrado em um mesmo lugar (linhas 16–31), tornando o sistema como um todo mais simples e legível. A Figura 7 mostra graficamente o resultado do código acima no sistema inteiro.

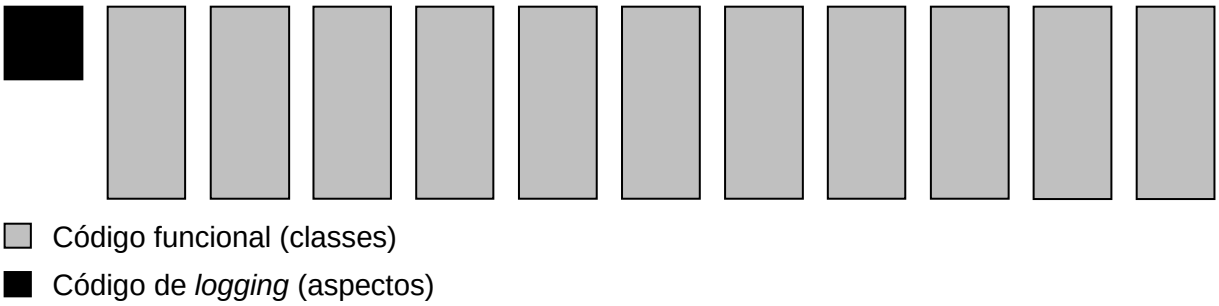


Figura 7 – Representação gráfica do código de *logging* separado das classes.
Fonte: O autor.

2.3 FRAMEWORKS ORIENTADOS A ASPECTOS

Com o passar do tempo, o foco da comunidade de orientação a aspectos vem mudando do projeto de linguagem e combinador específicos para cada aspecto para uma linguagem genérica baseada em poucos e bem definidos tipos de entrecortes[5] (*crosscuts*) [VAN01]. Com isso, todo o esforço de projeto de linguagem e combinador pode ser feito uma única vez, embora a implementação dos aspectos tenha se tornado mais difícil por exigir maior programação. Como resultado, o reúso do código dos aspectos tornou-se mais uma área de interesse para pesquisas.

A maneira como os aspectos são implementados pode variar de acordo com as características dos entrecortes entre a aplicação e o código de aspectos. Por um lado, os aspectos podem ser usados para adicionar código de forma invasiva e ortogonal a uma aplicação. Nesse caso a aplicação é independente dos aspectos, e permanece funcional se estes forem retirados. Existem poucos interesses transversais que podem ser descritos dessa forma, sendo o aspecto de *debugging* o exemplo mais conhecido. No outro extremo, o código dos

[5] Termos de orientação a aspectos seguindo a tradução definida no WASP'04

aspectos pode entrecortar profundamente uma aplicação. Isso acontece quando o estado, a estrutura e/ou a lógica dessa aplicação influenciam o código dos aspectos de tal modo que os aspectos ficam aplicáveis somente no contexto da própria aplicação. Dessa maneira, os aspectos são uma parte integral da aplicação, e ainda, uma aplicação pode ser composta de uma coleção de aspectos junto a uma estrutura principal da qual eles são baseados.

Entre os dois extremos, existem aspectos que não são tão ortogonais e nem tão dependentes em relação a uma determinada aplicação. Esses aspectos geralmente atingem interesses não-funcionais que necessitam de certa cooperação da aplicação para funcionarem. Além disso, em uma aplicação geralmente é necessário combinar o uso de diferentes propriedades. Ao invés de misturar todos os aspectos e gerar um caos, a utilização de uma estrutura genérica que os incorpore de forma coordenada gera melhores resultados, além da possibilidade de reutilização desta estrutura em diversas aplicações. É aí que entra em cena a tecnologia de frameworks. O grande desafio para os programadores de aspectos é desenhar uma solução de forma a ser possível combinar uma especificação geral com uma aplicação especializada.

Para generalizar os aspectos na forma de um framework, é necessária a identificação de quais os pontos em que a personalização dos aspectos é desejável, ou seja, os pontos de extensão, aquilo que será adaptado, desenvolvido e/ou parametrizado para concretizar a aplicação final. Primeiramente, é desejável permitir variações nos lugares onde os aspectos devem ser aplicados, ou seja, os pontos de junção da aplicação que os aspectos devem entrecortar. Para isso, podem ser utilizados conjuntos de junção abstratos, que serão concretizados durante o processo de reúso do framework. Após, quando é desejado um funcionamento intercambiável, a implementação da funcionalidade pode ser extraída do código do aspecto para algum tipo de mecanismo abstrato.

Boa parte da comunidade envolvida com aspectos considera como “framework orientado a aspectos” qualquer alternativa à linguagem AspectJ, ou seja, qualquer linguagem orientada a aspectos e o respectivo combinador desta linguagem com alguma linguagem de procedimentos generalizados[6] (Java, Pascal, C/C++, etc.). Nessa linha de pensamento, esses “frameworks” fornecem abstrações para a criação de aspectos, pontos de junção, conjuntos de junção e adendos, podendo oferecer algumas outras funcionalidades. Como exemplos dessa linha de pensamento podem ser citados Spring AOP [SPR02], JBoss AOP [JBO03] e AspectWerkz [ASP02]; basicamente, todos esses “frameworks” permitem a programação de

[6] Linguagens que possuem chamadas de procedimentos como único mecanismo de composição [KIC97a].

aspectos com o uso de alguma linguagem específica e a sua combinação com o código Java, em tempo de execução ou em tempo de compilação.

Neste trabalho, é considerado framework orientado a aspectos (FOA) o sistema semicompleto que encapsula o comportamento de determinado domínio, utilizando-se de orientação a aspectos, e que deve passar por um processo de reúso para a concretização de uma aplicação final, obtendo-se modularidade, reusabilidade, extensibilidade e inversão de controle [FAY99a][FAY99b][FAY99c]. A arquitetura de um FOA possui uma parte fixa e outra variável: essa parte variável precisa ser adaptada a fim de realizar a composição do FOA com uma aplicação existente ou de gerar uma nova aplicação [CAM05]. A adaptação geralmente envolve a concretização de mecanismos de composição abstratos, para os aspectos, e a concretização de classes e métodos abstratos, para as classes.

Do ponto de vista estrutural, um FOA é um conjunto de unidades básicas da POA (aspectos) e, opcionalmente, unidades básicas da POO (classes) (Figura 8). Isso significa que um FOA pode ser composto exclusivamente de aspectos, e, embora não seja comum, pode ocorrer em situações especiais [CAM05]. Esse conjunto de aspectos e classes representa o projeto abstrato de soluções para uma família de problemas relacionados [JOH88].

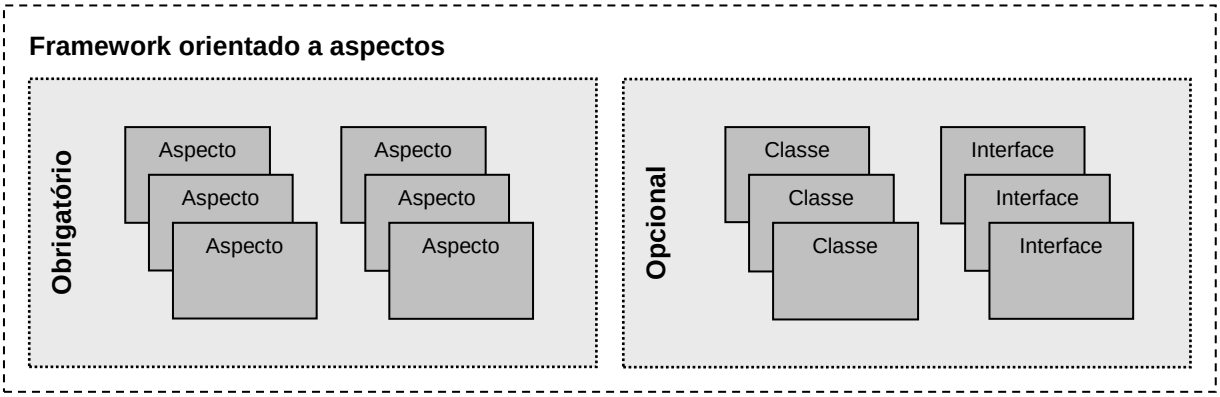


Figura 8 – Estrutura de um framework orientado a aspectos (FOA).
Fonte: O autor.

2.3.1 Vantagens e Desvantagens

Frameworks orientados a aspectos tendem a ser melhor modularizados e mais extensíveis se comparados aos frameworks orientados a objetos, pois possuem uma melhor organização de código e separação dos interesses envolvidos em sua estrutura, característica esta própria da orientação a aspectos. Possuem também maior reusabilidade, pois,

dependendo da sua natureza, podem ser utilizados em diferentes domínios, acoplados a aplicações já existentes ou a outros frameworks [CAM05]. Isso os diferencia dos FOOs, em que a aplicação é construída a partir do framework (ou da integração de vários frameworks distintos).

Contudo, essa maior abrangência de reúso também ocasiona alguns problemas. Assim como um aspecto, um FOA pode entrecortar um ponto da aplicação e modificar totalmente a semântica daquele ponto de forma intrusiva (por exemplo, substituindo o código de um método pelo código de um adendo). A mesma situação pode acontecer quando compondo um FOA com outro FOA, no qual um altera a semântica do outro. Isso exige mais cuidado por parte do reutilizador, para que situações inconsistentes não venham a ocorrer.

Deste modo, o entendimento de um FOA e do seu reúso também demanda mais esforço por parte do reutilizador, possivelmente aumentando a curva de aprendizado. O projeto e o desenvolvimento de FOAs também exigem mais esforço por parte do desenvolvedor do framework para tentar manter sua extensibilidade e restringir a sua atuação [CAM04a], causando impacto no tempo total de desenvolvimento do framework.

A eficiência de um FOA depende muito da forma como os aspectos são combinados com o código-base. Combinadores em tempo de compilação conseguem criar códigos praticamente tão eficientes quanto códigos entrelaçados, não influindo significativamente no desempenho do framework. Já combinadores em tempo de execução geralmente utilizam reflexão e camadas adicionais de monitoramento da execução, causando perdas de desempenho em relação a um código entrelaçado.

2.3.2 Classificação

Segundo Camargo e Masiero [CAM05], existem dois tipos de FOAs quanto à natureza: frameworks transversais e frameworks de aplicação orientados a aspectos.

Um framework transversal (FT) (*crosscutting framework*) é um FOA que possui mecanismos de composição abstratos e variabilidades correspondentes a um único interesse transversal, como, por exemplo, persistência, distribuição, segurança e regras de negócio. A principal característica desse tipo de FOA é a necessidade de acoplamento com algum código-base existente, isto é, sua instanciação somente não produz uma aplicação. Isso se deve pela natureza de um aspecto ser uma estrutura dependente: o código contido nos adendos de um

aspecto deve obrigatoriamente ser composto com um código-base previamente existente. Sendo assim, o processo de reúso possui duas etapas semanticamente distintas: instanciação e composição. Esse processo será detalhado na Subseção 2.3.4.

Do ponto de vista da aplicação, não ocorre a inversão de controle com o uso de FTs, pois é esta que determina o fluxo principal de execução. Porém, do ponto de vista interno do FT a inversão de controle continua ocorrendo, pois, assim como os demais frameworks, ele possui código genérico que chama métodos abstratos implementados na aplicação final, e o desenvolvedor de aplicações continua sem a responsabilidade de chamar os métodos do framework. Pode-se dizer assim que o fluxo de controle principal é da aplicação, mas dentro dos FTs o fluxo de controle é deles. Essa é uma característica que os difere das bibliotecas de classes, pois quando estas são utilizadas o fluxo de controle principal é sempre da aplicação. Nesse sentido, os FTs são similares aos *framelets* ([FAY99c], pp. 379-393), pois não assumem o controle da aplicação, têm interface de composição simples e bem definida, e geralmente têm um número pequeno de unidades de programação (classes e aspectos).

Um framework de aplicação orientado a aspectos (FAOA) (*aspect-oriented application framework*) é um FOA que implementa uma arquitetura genérica para um domínio, contendo classes e aspectos (para interesses transversais) de forma integrada, e o seu reúso produz uma aplicação deste domínio. Os FAOA são bastante parecidos com os frameworks de aplicação OO (frameworks de domínio específico) em relação ao seu propósito, sendo que a diferença principal entre ambos é arquitetural: os FAOAs utilizam classes e aspectos, concretos e abstratos, para implementar partes variáveis, e que serão concretizados durante o processo de reúso. Esse processo possuirá a etapa de instanciação, mas poderá ou não ter a etapa de composição. Nesse tipo de framework o princípio de inversão de controle continua válido.

A classificação mostrada nesta subseção é válida para frameworks caixa branca. Quando o framework é caixa preta, não importa como foi projetada a arquitetura da parte variável, pois todas as variabilidades já foram concretizadas e trata-se apenas de um processo de escolha de funcionalidades que geralmente é feito com auxílio automatizado.

2.3.3 Implementação de Variabilidades

A POA consiste de uma linguagem orientada a aspectos que captura e codifica somente os interesses transversais (aspectos), e também de uma linguagem de componentes para o

código funcional, ou seja, a POA não substitui outros paradigmas de programação (p.ex. POO) mas sim os complementa. Portanto, um FOA utiliza-se de orientação a objetos para codificar sua parte funcional, e de orientação a aspectos para sua parte transversal. Deste modo, a implementação das variabilidades dos FOAs pode utilizar todas as técnicas existentes da POO vistas na subseção 2.1.3, como herança, redefinição de métodos, interfaces e classes abstratas, padrões de projeto, seleção de classes e atribuição de valores.

Além disso, alguns desses conceitos de orientação a objetos foram trazidos para a orientação a aspectos, como herança de aspectos e aspectos abstratos, e também podem ser utilizados na construção de variabilidades. Mas o principal conceito de variabilidade introduzido pela POA são os conjuntos de junção abstratos [CAM04a]. Esses conjuntos de junção devem ser concretizados durante o processo de reúso, informando quais os pontos da aplicação (chamadas de métodos, construtores, tratamento de exceções, etc.) devem ser interceptados para que os adendos sejam executados. Isso permite a construção de adendos que irão atuar em pontos de junção desconhecidos no momento da construção do framework. Um problema com relação a esse mecanismo de extensão é a abrangência de atuação dos adendos associados a conjuntos de junção abstratos, já que estes podem ser concretizados de inúmeras formas, interceptando praticamente qualquer ponto de execução da aplicação. Para contornar esse problema e também implementar variabilidades, podem ser utilizados idiomas para AspectJ [HAN03], que fornecem soluções de implementação reusáveis em diversos domínios e situações.

Todos esses conceitos de variabilidade estão, de alguma forma, ligados às linguagem de aspectos e objetos utilizadas: por exemplo, herança múltipla, que não existe em algumas linguagens orientadas a objetos. Assim, dependendo das linguagens utilizadas na construção do framework, algumas formas de construção de variabilidades não estarão disponíveis, ou terão que ser construídas de uma forma diferente da habitual.

2.3.4 Processo de Reúso

O reúso de um FOA geralmente é mais abrangente do que de um FOO convencional, já que pode ser utilizado em diferentes domínios. Ele pode ser acoplado a aplicações existentes ou em desenvolvimento, a FOOs e a outros FOAs. Como já mencionado, o processo de reúso de um FOA possui duas etapas semanticamente distintas: instanciação e composição.

A instanciação é o processo convencional de reúso dos FOOs tradicionais e consiste em especializar e adaptar o código que foi especialmente projetado para isso (pontos de extensão). É durante a instanciação que ocorre a concretização dos pontos abstratos, a escolha de funcionalidades alternativas e/ou a implementação de novas funcionalidades. Isso é feito geralmente com a redefinição de métodos que retornam valores específicos da aplicação.

A etapa de composição por sua vez, consiste em duas atividades: identificação dos pontos de junção apropriados e fornecimento de regras de composição. A primeira atividade consiste em identificar no código-base os pontos de junção adequados ao acoplamento da funcionalidade do FOA, baseada nas "alternativas de composição" que o framework disponibiliza. É interessante que os FOAs sejam projetados com alternativas de composição, principalmente aqueles que necessitam de dados da aplicação em seu processamento, pois aumentam as chances de acoplamento com códigos-base previamente desenvolvidos, além de diminuir a complexidade das regras de composição que precisam ser fornecidas. No caso de um novo desenvolvimento, o código-base já pode ser projetado com vistas ao acoplamento que será feito; contudo, esse "desenvolvimento orientado às alternativas de composição" pode tornar o código-base confuso e difícil de manter, pois é possível que pontos de junção fictícios tenham que ser criados apenas para o acoplamento. Por exemplo, pode haver a necessidade da criação de métodos adicionais "falsos", que contenham as características necessárias à composição e apenas invoquem os métodos normais, sem nenhuma outra utilidade. A segunda atividade da etapa de composição consiste em fornecer regras que unam as variabilidades escolhidas e/ou implementadas do framework com o código-base, e, para isso, tarefas orientadas a aspectos devem ser realizadas, como, por exemplo, a concretização de um mecanismo de composição abstrato. Em alguns casos, a etapa de composição depende de informações que são determinadas na etapa de instanciação, o que determina uma ordem de realização: primeiro a instanciação e depois a composição. Mas essa dependência pode não existir, permitindo que ambas as etapas ocorram em qualquer ordem ou em paralelo.

Embora essas duas etapas sejam semanticamente distintas, sua separação "física" pode não existir durante o processo de reúso, pois isso depende do projeto do framework e da linguagem orientada a aspectos utilizada. O ideal é que o projeto do framework seja elaborado procurando separar as duas etapas o máximo possível, para que sua separação semântica continue existindo fisicamente. Algumas abordagens orientadas a aspectos facilitam essa tarefa, pois tendem a separar as regras de composição do comportamento transversal [TAR01][JAM03]; já outras, como AspectJ, que mantém as regras de composição no mesmo módulo do comportamento transversal, exigirão um maior cuidado do projetista.

O processo de reúso de um FOA possui três formas que muitas vezes é determinada pela natureza (FT ou FAOA) do framework: 1) somente instanciação, 2) somente composição, e 3) instanciação e composição. A primeira forma ocorre somente com FAOAs porque estes já possuem o código-base incluído em suas estruturas, não havendo a necessidade de acoplá-los a nenhum outro código. A maior parte de seus interesses transversais já está codificada internamente, bem como suas regras de composição. A segunda forma ocorre somente com FTs de uma única funcionalidade, que não necessitam ser adaptados. Um bom exemplo é um FT de rastreamento com a única funcionalidade de imprimir informações no console de saída padrão, que possui somente a fase de composição, a fim de informar os pontos do código base que seriam rastreados. Frameworks desse tipo são muito simples e muito limitados em suas capacidades de reúso. A terceira forma é mais complexa pois possui ambas as etapas de instanciação e de composição. Essa forma ocorre com mais freqüência em FTs adaptáveis, mas também pode ocorrer em FAOAs. Um exemplo é um FT de persistência, no qual suas variabilidades estão relacionadas ao mecanismo de persistência (arquivos, banco de dados, memória) e as devidas configurações (nome de diretórios, conexão ao banco de dados, etc.). O reúso desse framework consiste em determinar o mecanismo de persistência e sua configuração na etapa de instanciação e na composição da instância do framework com o código-base que deve ser persistido (Figura 9).

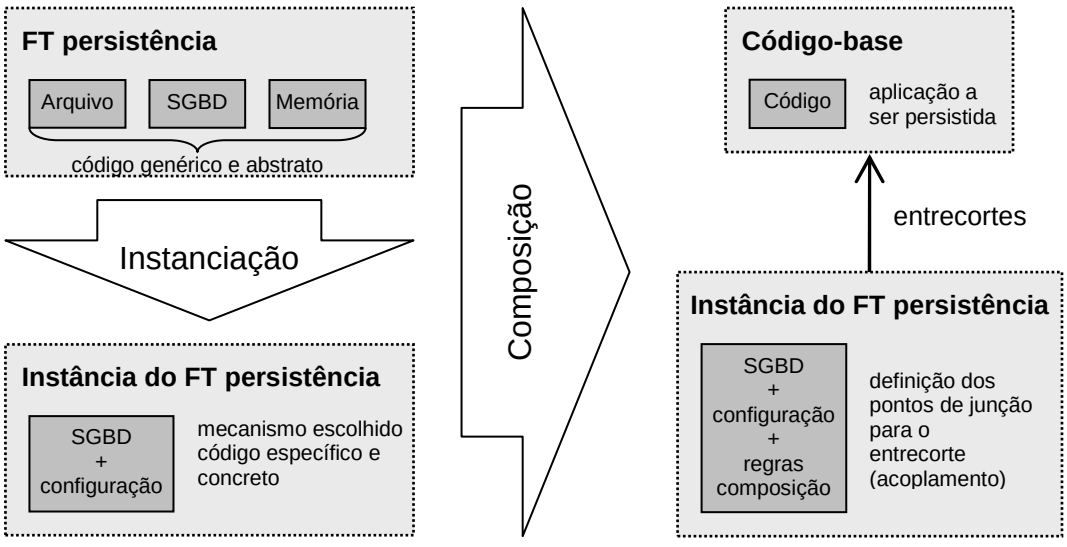


Figura 9 – Processo de reúso para um FT de persistência.
Fonte: O autor.

2.3.4.1 Documentação

Todas as abordagens de documentação mostradas na Subseção 2.1.4.1 foram projetadas com foco em frameworks orientados a objetos. Especificamente para FOAs foi proposta a abordagem UML-AOF [CAM04b]. A UML-AOF estende a UML-F [FON99] apresentada anteriormente a fim de fornecer a identificação de pontos de extensão de FOAs em seus diagramas de classes/aspectos. Ela fornece construções para diferenciar aspectos e classes da aplicação dos aspectos e classes do framework e para identificar métodos e conjuntos de junção envolvidos em um ponto de extensão. Como seu foco é apenas a documentação do framework, a identificação dos pontos de extensão carece de informações mais precisas sobre o que deverá ser feito com os mesmos durante o processo de reúso. A abordagem também carece de algum mecanismo para a especificação das atividades envolvidas no processo de reúso.

2.4 CONCLUSÕES

As técnicas de reúso de software evoluíram significativamente desde a criação das bibliotecas de procedimentos. Uma das técnicas mais difundidas atualmente é o desenvolvimento e uso de frameworks, que procuram reaproveitar tanto código quanto projeto, permitindo a geração de sistemas inteiros de uma forma muito rápida. O uso de separação de interesses nos frameworks ganhou força com o advento da programação orientada a aspectos, dando origem aos frameworks orientados a aspectos (FOAs).

Embora aspectos facilitem a organização do código, evitando espalhamento e entrelaçamento, também fazem com que o processo de reúso de um FOA seja mais complexo que o de um FOO, pois, além da instanciação, pode possuir uma etapa adicional de composição; isso se deve ao fato de um aspecto ser uma estrutura dependente, que deve ser composta com um código funcional para executar. Além disso, a implementação de variabilidades dos FOAs pode utilizar todas as técnicas de orientação a objetos, já que a POA é utilizada em conjunto com a POO (e não a substituindo), acrescidas das novas técnicas que a orientação a aspectos introduziu.

Neste contexto, a documentação do processo de reúso dos FOAs é necessária para facilitar a compreensão do processo e diminuir a quantidade de erros gerada pelo mesmo. De

preferência, a técnica de documentação deve ser formal, para evitar ambigüidade, e capaz de ser processada por um computador; isso facilita a detecção de problemas e aumenta o controle do desenvolvedor de aplicações sobre o processo de reúso. Existem diversas abordagens para facilitar a compreensão dos pontos de extensão e de suas formas de uso, mas praticamente todas foram criadas com foco em FOOs, sendo que a grande maioria ou são documentos textuais estruturados ou são fortemente amarradas ao domínio ou linguagem de programação do framework. A que mais se destaca é a abordagem RDL, que utiliza uma linguagem de processos (RDL) para capturar os passos de instanciação e um ambiente (xFIT) para execução de programas de instanciação construídos nesta linguagem, sistematizando assim o processo de reúso de FOOs.

Como capturar variabilidades orientadas a objetos é um requisito para uma abordagem de sistematização do reúso de FOAs, neste trabalho a abordagem RDL foi utilizada como base para a construção de uma nova abordagem com foco em FOAs.

3 DOCUMENTAÇÃO DE FRAMEWORKS ORIENTADOS A ASPECTOS

Para que um framework possa ser reutilizado no desenvolvimento de uma aplicação, é de fundamental importância que seus reutilizadores – os desenvolvedores de aplicação – compreendam o framework: seu objetivo, sua estrutura, seus pontos de extensão e seu processo de reúso. Sem isso, os desenvolvedores de aplicação não utilizarão esse framework em suas aplicações, desperdiçando todo o esforço empregue no seu desenvolvimento. Essa compreensão do framework pode ser traduzida nas seguintes perguntas:

- **Para que serve?** – qual o objetivo do framework: domínio de aplicações geradas, função de infra-estrutura e/ou integração realizadas, etc.
- **Como ele é?** – qual o projeto (estrutura, comportamento e arquitetura) do framework (classes, métodos, atributos, aspectos, conjuntos de junção, adendos, bem como as interações entre estes elementos).
- **Quais são as possibilidades de reúso?** – identificação dos pontos de extensão, suas características, funcionalidades e restrições.
- **Como fazer o reúso?** – especificação da seqüência de tarefas que devem ser realizadas para o correto preenchimento dos pontos de extensão.

As respostas para essas perguntas têm como ponto de partida o entendimento de uma documentação capaz de representar todas essas informações. Ocorre que, normalmente, essa documentação é feita de uma forma não-estruturada e informal; embora o objetivo do framework possa ser perfeitamente expresso dessa forma, o mesmo não acontece com o projeto, os pontos de extensão e o processo de reúso, exigindo um grande esforço de aprendizado por parte do reutilizador. Esse esforço de aprendizado poderia ser drasticamente reduzido com o uso de uma forma padronizada de representação de conhecimento, para que as especificações pudessem ser compreendidas, comparadas e utilizadas de forma sistemática.

Atualmente existe uma linguagem padrão para a representação do projeto de sistemas orientados a objetos: a UML [OMG06a]. Ela é capaz de representar todos os elementos de um sistema orientado a objetos, como classes, interfaces, métodos, atributos, bem como os relacionamentos entre esses elementos, e o comportamento, o fluxo de dados e a arquitetura do sistema, conseguindo desse modo modelar perfeitamente o projeto de um FOO. Contudo, o foco da UML é em orientação a objetos, e até sua versão atual (2.0) não possui suporte a programação orientada a aspectos. Visando a modelagem de sistemas orientados a aspectos, diversas abordagens estendendo a UML foram propostas ([BAR04], [CHA04], [CLA01],

[HAN04], [STE02], [SUZ99], entre outras), mas nenhuma foi adotada massivamente como padrão. Como a modelagem orientada a aspectos é fundamental para a correta representação do projeto de um FOA, ela será vista em maiores detalhes na Subseção 3.1.

Mas somente a representação do projeto de uma framework não é suficiente. É preciso que seus pontos de extensão sejam facilmente identificáveis, para que o reutilizador compreenda suas características, funcionalidades e restrições, e até mesmo ter noção das capacidades do framework. Uma forma natural de identificar os pontos de extensão no projeto é estender a UML de modo a capturar as características relevantes ao processo de reúso, como faz a UML-FI [OLI01][OLI04] para FOOs. Este trabalho apresenta uma extensão da UML-FI que incorpora as novas características introduzidas pela POA no desenvolvimento de pontos de extensão, e que será mostrada em detalhes na Subseção 3.2.

Como já dito anteriormente, além da identificação dos pontos de extensão é necessária também a especificação da seqüência de tarefas que devem ser realizadas, a fim de que estes pontos de extensão sejam corretamente preenchidos. Para isso, este trabalho propõe uma linguagem específica, a ser abordada em detalhes no Capítulo 4.

3.1 MODELAGEM ORIENTADA A ASPECTOS

O objetivo principal da modelagem orientada a aspectos é capturar adequadamente os interesses transversais com uma linguagem de modelagem que forneça notação e conceitos precisos e específicos para representar os elementos da orientação a aspectos de forma padronizada. A modelagem orientada a aspectos deve permitir ao modelador escolher e explicitamente capturar qualquer tipo de aspectos no domínio do problema, e qualquer tipo de dependência entre aspectos e código funcional.

Embora a POA possa coexistir com qualquer paradigma de programação procedural [KIC97a], tanto sua utilização quanto sua pesquisa vêm sendo feitas em quase sua totalidade em conjunto com a POO. Portanto, nada mais natural que a modelagem de aspectos seja relacionada à UML, o que é ideal para a modelagem de FOAs, já que estes podem utilizar (e geralmente utilizam) orientação a objetos internamente. Mas como a UML, até sua versão atual (2.0), não possui suporte a aspectos, foram propostas diversas abordagens estendendo-a de forma a introduzir os conceitos de orientação a aspectos, mas nenhuma até agora se tornou padrão. Essas abordagens são baseadas ou em extensão leve (sem alteração do metamodelo da

UML) [BAR04][STE02][SUZ99] ou em extensão pesada (com alterações no metamodelo da UML) [CHA04][CLA01][HAN04].

Basicamente, as abordagens de extensão leve definem um conjunto de estereótipos (*stereotypes*) e valores identificados (*tagged values*) que servem para derivar os conceitos da orientação a aspectos das metaclasses da UML. As principais vantagens de abordagens desse tipo são: a facilidade de compreensão, pois são mais intuitivas, lidam com conceitos já assimilados pelos desenvolvedores (metaclasses padrão da UML); a capacidade de serem processadas pelas atuais ferramentas CASE do mercado; e a capacidade dos modelos gerados serem transformados no formato XMI [OMG06b]. Mas o uso das metaclasses UML para capturar os conceitos da POA possui certas limitações, não sendo capaz de representar completamente alguns conceitos, como declarações intertipo e operações de entrecorte, e trazendo algumas incoerências semânticas, como a obrigatoriedade de atribuição de nome a adendos (que não possuem identificação), quando estes são derivados de operações. Para contornar esses problemas, algumas dessas abordagens, como a AODM [STE02], introduzem notações não padronizadas pela UML para representar alguns conceitos da POA, porém ocasionando a perda de suporte das ferramentas CASE atuais para tais conceitos.

As abordagens que alteram o metamodelo conseguem representar mais claramente e corretamente os conceitos de POA, pois adicionam novos elementos ao metamodelo com as características necessárias e com símbolos diferenciados, como a Theme/UML [CLA01], que utiliza padrões de composição (*composition patterns*) para modelar tanto o comportamento transversal quanto o sistema como um todo, ou a aSideML [CHA04], que introduz novos elementos e diagramas para representar aspectos e comportamentos transversais. Mas pelo fato dessas abordagens alterarem o metamodelo, seus diagramas não conseguem ser criados e utilizados pelas ferramentas CASE atuais e por tecnologias relacionadas ao UML, como XMI, pois estas possuem suporte apenas ao metamodelo padrão da UML. Além disso, algumas abordagens ainda não representam completamente todos os conceitos de POA, como a aSideML [CHA04], que não suporta herança entre aspectos nem a declaração explícita de conjuntos de junção, os dois principais mecanismos de construção de variabilidades de FOAs.

O framework de segurança apresentado em [CAM04a] é utilizado para ilustrar alguns exemplos de diagramas utilizando algumas abordagens de modelagem orientada a aspectos. A Figura 10 exibe o modelo do FOA como originalmente proposto em [CAM04a], segundo o qual o autor utiliza uma notação intuitiva própria com estereótipos. É importante perceber o uso de notas em linguagem natural para identificar e explicar os pontos de extensão, já que a linguagem de modelagem não oferece mecanismos padronizados para isso.

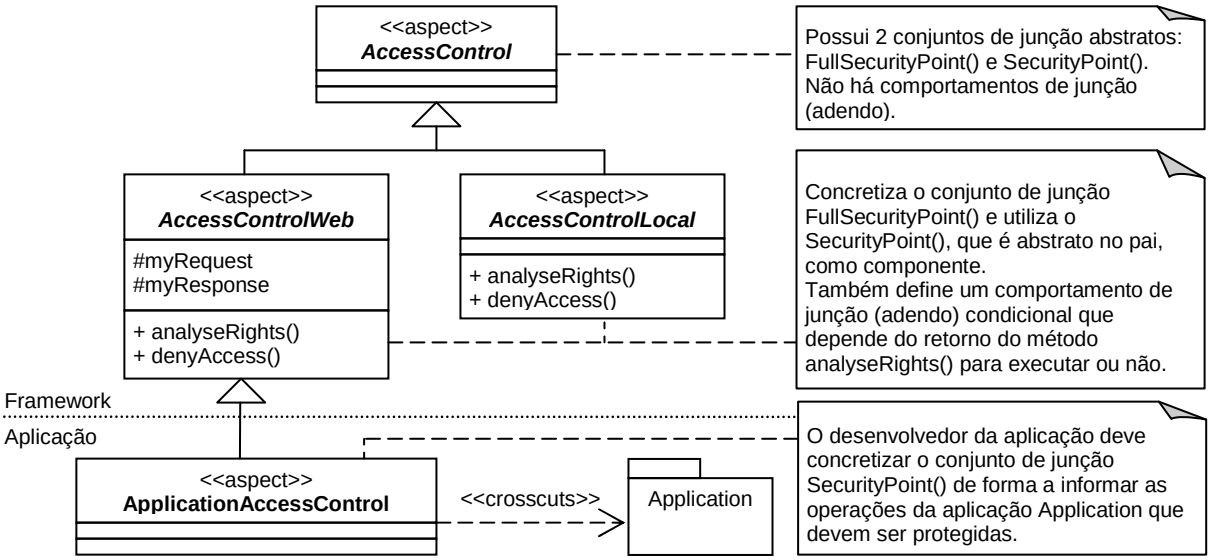


Figura 10 – Framework de segurança modelado originalmente por Camargo.
Fonte: Camargo [CAM04a].

A Figura 11 mostra o diagrama do mesmo framework modelado com a AODM [STE02]. Pode-se notar que foi atribuído um nome (identificador) "falso" ao adendo que atua sobre o conjunto de junção *FullSecurityPoint*, pois a representação dos adendos deriva diretamente de uma operação UML, que exige um identificador.

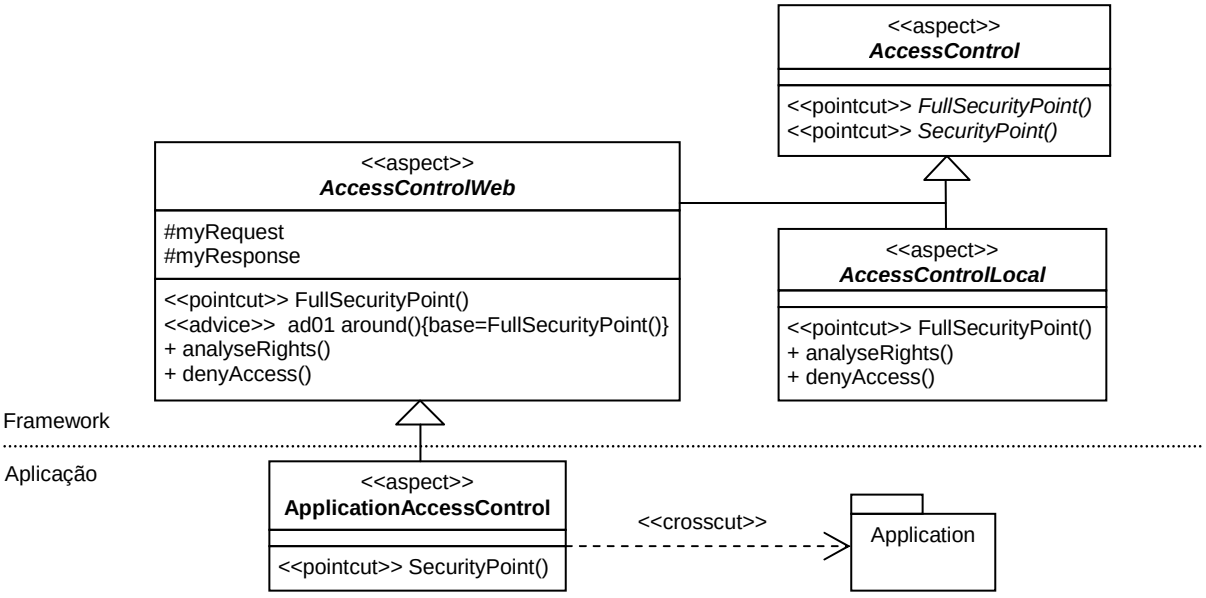


Figura 11 – Framework de segurança modelado com AODM.
Fonte: O autor.

A modelagem do framework com a abordagem aSideML proposta em [CHA04] é bem mais complicada, pois esta não suporta herança entre aspectos, deixada como trabalho futuro, nem aspectos abstratos e conjuntos de junção, sendo a parametrização dos aspectos o único mecanismo de reúso oferecido. Conseqüentemente, não é possível expressar o aspecto

abstrato *AccessControl*, que contém apenas dois conjuntos de junção abstratos, nem a hierarquia entre os aspectos. O modelo mostrado exibe somente o aspecto concreto da aplicação, obtido após o processo de reúso.

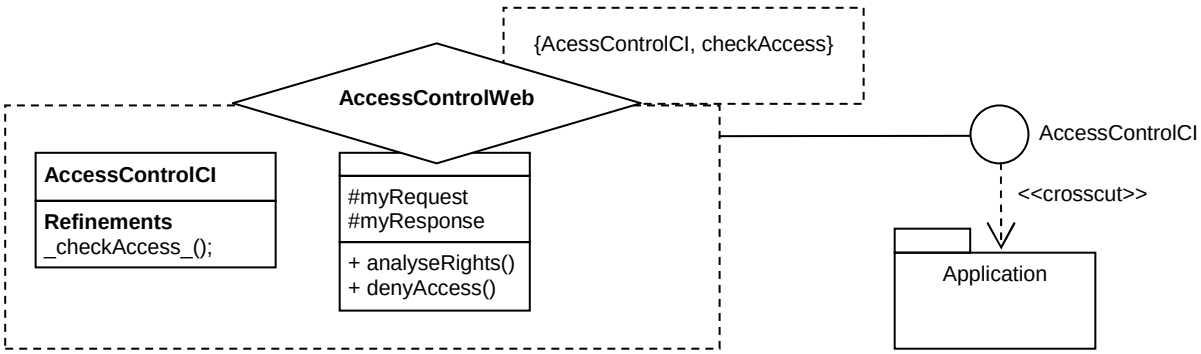


Figura 12 – Framework de segurança modelado com aSideML.
Fonte: O autor.

Como pode ser visto nos exemplos, a capacidade do modelo de um framework representar com exatidão sua estrutura, arquitetura e comportamento dependerá muito da linguagem de modelagem escolhida. Algumas abordagens podem contornar os problemas de modelagem de FOAs vistos acima, mas por sua vez podem gerar outros. Também vai influenciar nessa escolha o desejo ou não de suporte pelas atuais ferramentas CASE ou pelo formato XMI, que decidirá por extensões leves ou pesadas da UML. Enquanto nenhuma linguagem de modelagem orientada a aspectos for definida como padrão para a modelagem de FOAs, o ideal é que tecnologias relacionadas à documentação destes frameworks sejam construídas de forma independente da linguagem de modelagem, utilizando, por exemplo, o mecanismo de extensão leve da UML (estereótipos e valores identificados), permitindo assim a livre escolha por parte do modelador do framework.

Também fica evidente, nos exemplos mostrados, a falta de construções padronizadas para a identificação e caracterização dos pontos de extensão do framework de segurança, tanto que na Figura 10 o autor coloca essas informações informalmente no modelo, em linguagem natural dentro de notas da UML. De forma análoga às tecnologias para especificar as características de extensão e reúso dos FOOs em modelos UML, como a UML-F e a UML-FI, este trabalho propõe uma extensão que realiza estas tarefas para FOAs, apresentada na próxima subseção.

3.2 UML-AFR

A UML-AFR (*Aspect-oriented Framework Reuse*) complementa as linguagens de modelagem orientada a aspectos, fornecendo construções para a identificação e documentação dos pontos de extensão nos diagramas de classes/aspectos. Ela estende a UML-FI adicionando construções capazes de representar os novos tipos de pontos de extensão que a orientação a aspectos introduziu na construção de frameworks. Todas as construções da UML-AFR se baseiam em estereótipos e valores identificados – o mecanismo de extensão leve da UML – permitindo o seu uso em conjunto com qualquer abordagem de modelagem orientada a aspectos baseada em UML.

A UML-AFR expande o conceito de elemento reutilizável introduzido pela UML-FI, adicionando os novos tipos **aspecto reutilizável** e **conjunto de junção reutilizável**. Assim como nos demais tipos de elementos reutilizáveis (classe reutilizável, método reutilizável e atributo reutilizável[7]), seus elementos podem estar presentes ou não no projeto final, caracterizando a obrigatoriedade ou não do mesmo. Utilizando a notação UML, essa característica é representada pelo valor identificado ***presence*** igual a *optional*, no caso do elemento em questão – classe, método, atributo, aspecto ou conjunto de junção – ser opcional, ou igual ao valor padrão *mandatory*, que caracteriza a obrigatoriedade do mesmo no projeto final. A característica de determinado elemento ser opcional pode ser propagada:

- se uma classe é opcional, seus membros também são opcionais [OLI01][OLI04];
- se um atributo é opcional, a classe que modela este atributo é opcional, caso não seja referenciada por nenhum outro elemento obrigatório [OLI01][OLI04];
- se um método é opcional, as classes que modelam seus parâmetros são opcionais, caso não sejam referenciadas por nenhum outro elemento obrigatório [OLI01][OLI04];
- se um aspecto é opcional, seus conjuntos de junção também são opcionais.

A Figura 13 exibe graficamente essas regras de propagação da opcionalidade. Como o aspecto *Aspect1* é opcional, seu único conjunto de junção *InterceptPoint* também é, bem como o adendo associado a este conjunto de junção. A classe *Class0* propaga seu caráter opcional para todos os seus membros: o atributo *name* e os métodos *getName* e *setName*. Como o atributo *refClass2*, responsável pela associação entre *Class1* e *Class2*, é opcional, a classe que o modela, *Class2*, também será opcional caso não seja referenciada por nenhum

[7] Uma associação é vista como um atributo.

outro elemento obrigatório; se algum outro elemento obrigatório referenciar a classe *Class2*, esta será então obrigatória, pois a obrigatoriedade sempre será dominante. A mesma situação ocorre para o método *calculate* da classe *Class1* e a classe *Class3*.

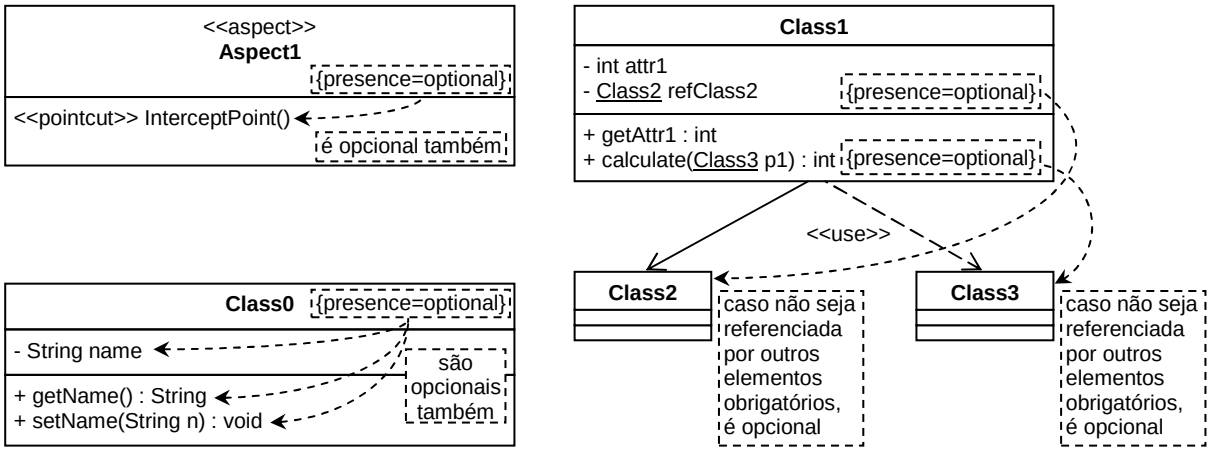


Figura 13 – Propagação da opcionalidade.
Fonte: O autor.

A UML-AFR também possui estereótipos para representar os tipos de reúso possíveis para classes, métodos e atributos reutilizáveis, herdados diretamente da UML-FI. Esses estereótipos identificam os pontos de extensão no modelo do framework, facilitando a localização destes pelo reutilizador e dando uma idéia geral do que deverá ser feito para o preenchimento dos mesmos durante o processo de reúso. Os estereótipos em questão são:

- **class_extension** – a classe reutilizável deve ser especializada durante o processo de reúso.
- **pattern_class_extension** – a classe reutilizável deve ser especializada durante o processo de reúso pelo uso de um padrão de projeto (*design pattern*).
- **select_class_extension** – uma das subclasses concretas da classe reutilizável deve ser escolhida durante o processo de reúso.
- **method_extension** – o método reutilizável deve ser redefinido na(s) subclasse(s) durante o processo de reúso.
- **pattern_method_extension** – o método reutilizável deve ser redefinido na(s) subclasse(s) durante o processo de reúso pelo uso de um padrão de projeto.
- **value_assignment_extension** – um valor válido deve obrigatoriamente ser atribuído ao atributo reutilizável durante o processo de reúso.
- **value_selection_extension** – um valor válido, selecionado de uma lista de valores possíveis, deve obrigatoriamente ser atribuído ao atributo reutilizável durante o processo de reúso.

Além desses estereótipos, que possuem foco voltado à orientação a objetos, novos estereótipos foram criados para identificar aspectos e conjuntos de junção reutilizáveis e suas formas de reúso nos diagramas de FOAs.

Assim como nas classes, a operação básica de reutilização de um aspecto é a sua especialização. A atividade de especialização de um aspecto tem como objetivo criar um novo aspecto de modo a representar as características específicas da aplicação, ao mesmo tempo reutilizando o código definido no aspecto original. Por exemplo, no framework de segurança exibido anteriormente, o aspecto *AccessControlWeb* precisou ser especializado para capturar características específicas da aplicação (Figura 14). Para representar essa situação o aspecto deve ser decorado com o estereótipo **aspect_extension**.

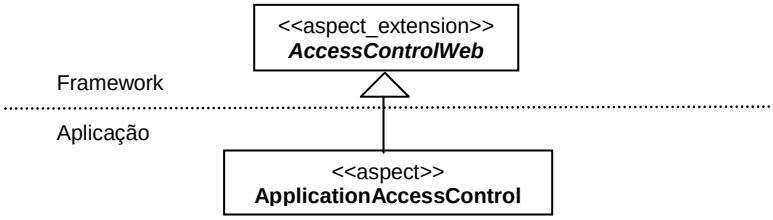


Figura 14 – Especificação do aspecto *AccessControlWeb* para extensão.
Fonte: O autor.

Outra possibilidade é quando o próprio desenvolvedor do framework implementa diversas especializações de um aspecto mais genérico, uma para cada situação, visando facilitar o trabalho do desenvolvedor. Ao invés de criar uma nova implementação, o reutilizador tem a opção de selecionar uma dessas implementações concretas para ser utilizada na aplicação. Esse conceito é representado no modelo decorando-se o aspecto com o estereótipo **select_aspect_extension** (Figura 15).

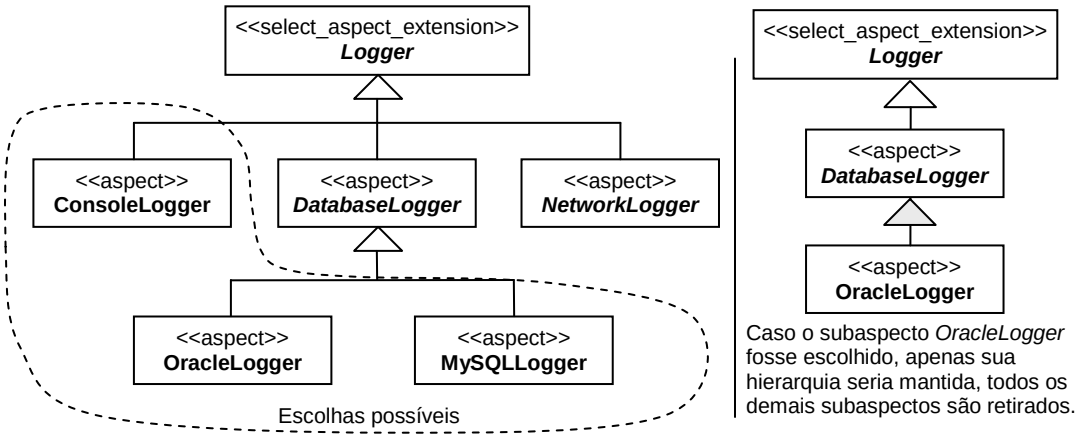


Figura 15 – Especificação do aspecto *Logger* para extensão por seleção.
Fonte: O autor.

O outro elemento importante para o processo de reúso de FOAs é o conjunto de junção. A forma de reúso de um conjunto de junção é a redefinição, permitindo adicionar pontos de junção que devem ser entrecortados pelo código de seus adendos associados. A definição no modelo de um conjunto de junção que necessita ser redefinido durante o processo de reúso é feita com a sua decoração pelo estereótipo **pointcut_extension**. Um exemplo utilizando o framework de segurança é mostrado na Figura 16, no qual o conjunto de junção *SecurityPoint* deve ser estendido.

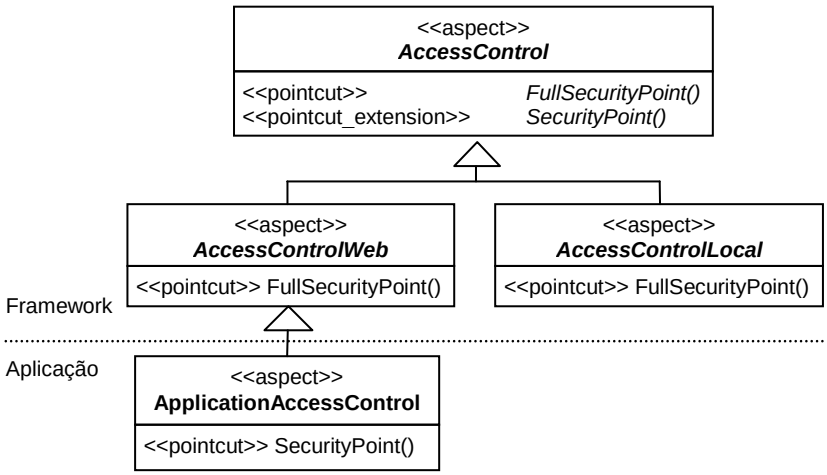


Figura 16 – Conjunto de junção *SecurityPoint* para extensão.
Fonte: O autor.

Com já mencionado, a UML-AFR estende a UML-FI, que por sua vez estende a UML. Para uma perfeita integração, é importante descrever a especificação da UML-AFR junto ao metamodelo da UML. Isso possibilita delimitar o seu escopo de aplicação, já que a UML-AFR introduz uma semântica especial quando utilizada.

Como a própria UML-FI foi construída com possibilidades de extensão, a UML-AFR apenas define novos elementos reutilizáveis – *ReusableAspect* e *ReusablePointcut* – a partir da metaclasse *ReusableElement*, com suas devidas características. A particularidade é que esses dois novos elementos atuam sobre metaclasses que, embora não existam no metamodelo padrão da UML, serão adicionadas pela linguagem de modelagem orientada a aspectos utilizada. Isso permite que a UML-AFR seja usada em conjunto com qualquer abordagem de modelagem baseada na UML que forneça construções para aspectos e conjuntos de junção, bastando que as metaclasses representando estas construções herdem, respectivamente, de *ReusableAspect* e *ReusablePointcut*.

O resultado da integração dos elementos da UML-AFR com o metamodelo da UML é apresentado no diagrama de classes da Figura 17.

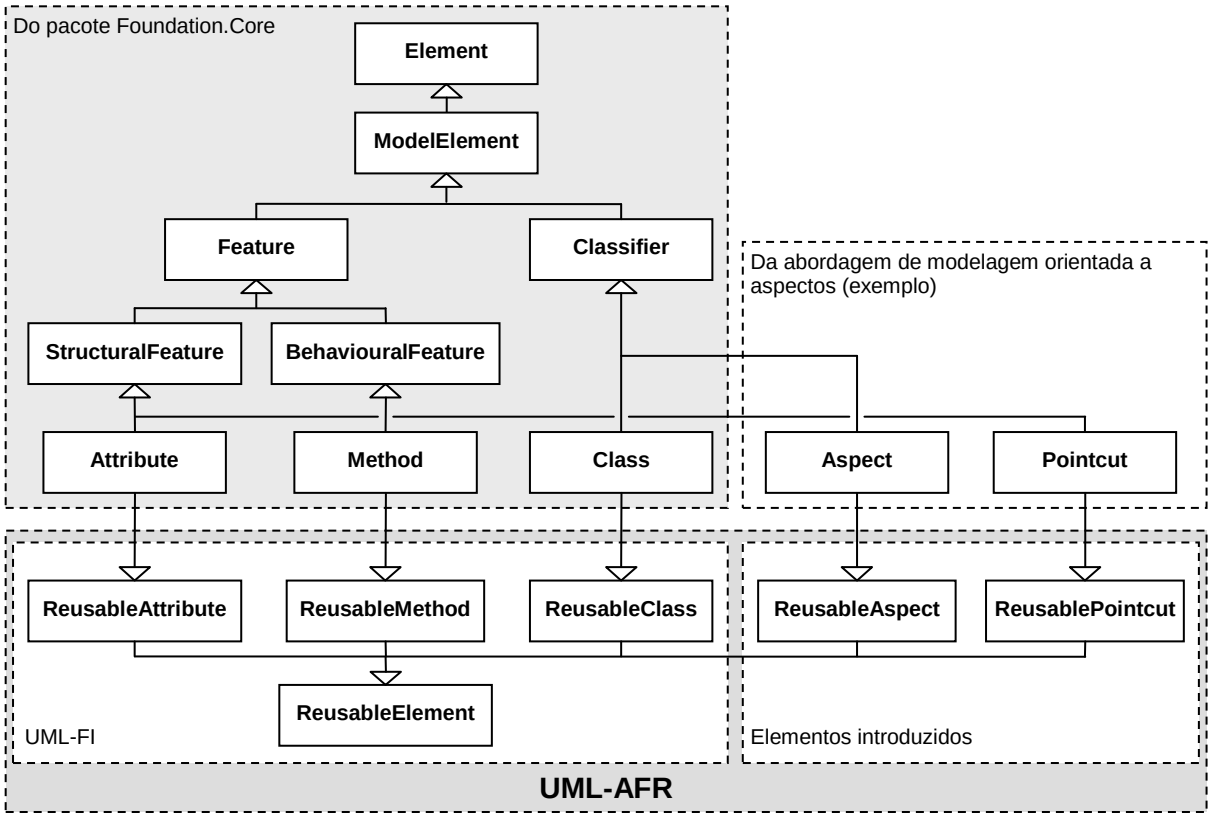


Figura 17 – Incorporação da UML-AFR ao metamodelo da UML.
Fonte: O autor.

O Quadro 5 exibe em detalhes cada um dos elementos da UML-AFR. Ao lado do nome segue uma breve descrição do elemento, seguido também da referência bibliográfica no caso dos elementos trazidos da UML-FI. Logo abaixo, são exibidos os possíveis estereótipos e valores identificados que podem ser usados no elemento em questão, com suas respectivas explicações em linguagem natural. Por fim, são mostradas as regras de formação de cada elemento que definem como a obrigatoriedade ou não do elemento afeta outros elementos relacionados. Cada regra é especificada também em OCL (*Object Constraint Language*), uma linguagem da UML para definição de pré-condições, pós-condições e restrições, permitindo assim garantir a integridade do modelo gerado.

ReusableElement	Raiz da hierarquia de elementos reutilizáveis. [OLI01][OLI04]	
Valor identificado	**presence** (valores possíveis: *optional* e *mandatory* [padrão]) Indica se o elemento é obrigatório ou opcional no projeto final.	
ReusableClass	Classe reutilizável. [OLI01][OLI04]	
Estereótipos	**class_extension**	requer especialização.
	pattern_class_extension	requer especialização por um padrão de projeto.
	select_class_extension	requer especialização via seleção de subclasse concreta.
Regra de formação	1) se a classe é opcional, seus membros também são opcionais.	

	`context Class`
	`inv: self.presence = optional implies self.allAttributes->forAll(presence = optional)`
	`inv: self.presence = optional implies self.allMethods->forAll(presence = optional)`	

ReusableMethod	Método reutilizável. [OLI01][OLI04]	
Estereótipos	**method_extension**	requer redefinição na(s) subclasse(s).
	pattern_method_extension	requer redefinição na(s) subclasse(s) via padrão de projeto.
Regras de formação	1) se o método é opcional, as classes que modelam seus parâmetros também são opcionais caso não exista nenhuma outra referência de obrigatoriedade a estas classes. 2) se o método é obrigatório, as classes que modelam seus parâmetros também são obrigatórias.	

	`context Method`
	`inv: (self.presence = optional and self.ownedParameter->`
	`forAll(type.allInstances.presence = optional)`
	`) implies self.ownedParameter->forAll(type.presence = optional)`
	`inv: self.presence = mandatory implies self.ownedParameter->forAll(type.presence=mandatory)`	

ReusableAttribute	Atributo reutilizável. [OLI01][OLI04]	
Estereótipos	**value_assignment_extension**	requer atribuição de um valor válido
	value_selection_extension	requer atribuição de um valor de uma faixa permitida.
Regra de formação	1) se o atributo é opcional, a classe que o modela também é opcional, caso não exista nenhuma outra referência de obrigatoriedade a esta classe. 2) se o atributo é obrigatório, a classe que modela este atributo também é obrigatória.	

	`context Attribute`
	`inv: (self.presence = optional and self.type.allInstances.presence = optional) implies`
	`self.type.presence = optional`
	`inv: self.presence = mandatory implies self.type.presence = mandatory`	

ReusableAspect	Aspecto reutilizável.	
Estereótipos	**aspect_extension**	requer especialização.
	select_aspect_extension	requer especialização via seleção de subaspecto concreto.
Regra de formação	1) se o aspecto é opcional, seus conjuntos de junção também são opcionais.	

	`context Aspect`
	`inv: self.presence = optional implies self.allFeatures->select(oclIsKindOf(Pointcut))->`
	`forAll(presence = optional)`	

ReusablePointcut	Conjunto de junção reutilizável	
Estereótipo	**pointcut_extension**	requer redefinição no(s) subaspecto(s).

Quadro 5 – Representação e regras de formação dos elementos reutilizáveis.
Fonte: Oliveira [OLI01][OLI04] e o autor.

3.3 CONSIDERAÇÕES FINAIS

A notação UML-AFR representa os pontos de extensão de um FOA sob o ponto de vista estático de um diagrama de classes/aspectos. Essa representação permite a rápida visualização, identificação e entendimento desses pontos de extensão presentes. Um ponto importante dessa notação é o uso de mecanismos de extensão leve da UML, permitindo a sua utilização com a abordagem de modelagem orientada a aspectos que mais convier ao desenvolvedor do framework, desde que esta abordagem seja baseada em UML e suporte construções para aspectos e conjuntos de junção.

Como exemplo do funcionamento da UML-AFR, a Figura 18 ilustra a modelagem do framework de segurança utilizando a UML-AFR em conjunto com a AODM. Nesse exemplo é possível notar como os estereótipos da UML-AFR identificam os pontos de extensão do framework em questão, mostrando que tanto o aspecto *AccessControlWeb* quanto o *AccessControlLocal* podem ser estendidos durante o reúso, e que nestes subaspectos deverá ser concretizado o conjunto de junção *SecurityPoint*, como indicado. O valor identificado *presence* também indica que a presença dos aspectos é opcional na aplicação final, pois a escolha de qual aspecto será estendido (ou até mesmo se nenhum, em caso de não haver a necessidade de segurança) depende dos requisitos da aplicação (se a aplicação é *web* ou é local).

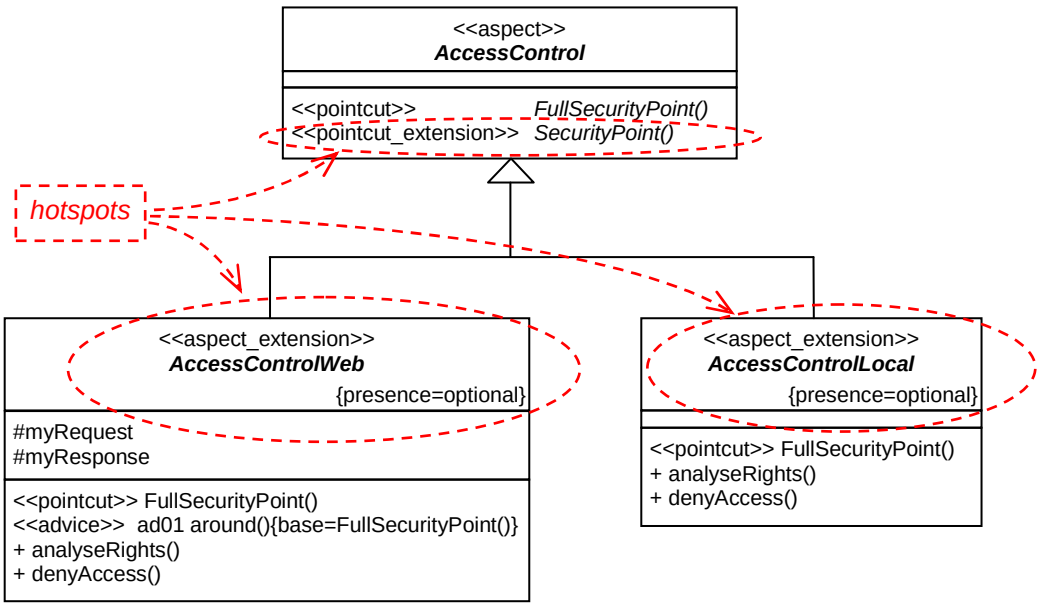


Figura 18 – Framework de segurança modelado com UML-AFR e AODM.
Fonte: O autor.

Entretanto, essa representação não está completa. A parte de declarações intertipo (introduções) foi deixada de fora nesta versão da notação por não haver um mecanismo de extensão genérico bem definido, tanto nas linguagens de POA quanto nas de modelagem, de para as mesmas; isso é deixado para uma nova versão da notação, quando tais mecanismos estiverem disponíveis. Além disso, alguns itens foram propositadamente omitidos para não prejudicar a legibilidade do modelo, como os valores envolvidos em uma seleção ou a identificação do padrão de projeto utilizado.

Outro ponto omitido na notação UML-AFR é a seqüência em que as atividades de reúso devem ocorrer. Especificar essa seqüência em um diagrama seria como programar um sistema inteiro de forma visual em diagramas de seqüência ou colaboração. Essa idéia possui um problema relativo à capacidade de tais diagramas conseguirem representar completamente as atividades de reúso, já que estes não são computacionalmente completos.

4 ESPECIFICAÇÃO DAS ATIVIDADES DE REÚSO

Além da documentação do projeto (estrutura, comportamento e arquitetura) e da identificação dos pontos de extensão, é necessário compreender como estes pontos devem ser preenchidos para que o framework seja corretamente reutilizado (instanciado e/ou composto com algum código), gerando assim a aplicação final. O preenchimento dos pontos de extensão é realizado por tarefas repetitivas como concretização de mecanismos de composição abstratos, adaptação de interfaces, redefinição de métodos, atribuição de valores e verificação de restrições. Como se trata de tarefas repetitivas, o uso de uma abordagem sistemática ajudaria a reduzir a quantidade de erros inerentes ao processo de reúso, acarretando em um projeto mais confiável da aplicação final.

Além disso, esse preenchimento deve seguir uma ordem pré-estabelecida pelo desenvolvedor do framework, uma vez que certos pontos de extensão podem depender de outros, além da possível dependência que pode existir entre as etapas do reúso (instanciação e composição). Por possuir total conhecimento do framework e de seu funcionamento, o desenvolvedor do framework deve ser capaz de transmitir este conhecimento, pela especificação da seqüência de atividades, parâmetros pertinentes e restrições necessárias, ao reutilizador (desenvolvedor da aplicação), de forma a auxiliá-lo a obter corretamente o projeto da aplicação final.

Se uma linguagem de programação capturasse todas essas atividades de reúso em instruções bem definidas, essa seqüência de atividades poderia ser determinada por um ou mais programas[8] de reúso. O uso de uma linguagem de programação evita ambigüidades e falhas de entendimento em comparação com a linguagem natural, que possui uma natureza ambígua e interpretativa: dois reutilizadores podem interpretar distintamente a mesma especificação em linguagem natural. Além disso, abre espaço para que esses programas de reúso possam ser executados por um computador, permitindo auxílio automatizado.

Este trabalho define uma linguagem de programação, denominada RDL+Aspects, que captura as possíveis atividades envolvidas no processo de reúso de FOAs em declarações bem definidas e independentes do domínio do framework. Seus comandos manipulam elementos no nível de projeto, realizando, em conjunto com a interação do reutilizador, a transformação do projeto do framework no projeto da aplicação. A realização do reúso na fase de projeto, além de manter os programas de reúso independentes da linguagem de programação do

[8] Programa é definido como uma seqüência de instruções capazes de serem processadas por um computador.

framework, permite uma melhor especificação da aplicação final antes da implementação, aumentando a correspondência entre projeto e implementação. Caso o reúso fosse feito somente na fase de implementação o projeto da aplicação final teria que ser refeito de modo a replicar as modificações causadas pelo reúso. Essa etapa adicional de engenharia reversa possivelmente aumentaria a taxa de erros gerados no processo de desenvolvimento da aplicação.

4.1 RDL+ASPECTS

A RDL+Aspects (*Reuse Description Language enhanced with Aspects*) possui mecanismos de abstração baseados em *Cookbooks* [KRA88], estendendo e revisando a linguagem RDL: esta foi escolhida por já possuir comandos que representam as atividades de reúso relacionadas a variabilidades orientadas a objetos, que também podem ocorrer no reúso de FOAs. A linguagem permite a construção de programas (*cookbooks*) tanto de instanciação quanto de composição que podem ser executados de forma manual ou assistida (com a ferramenta *Reuse Tool*, a ser vista no Capítulo 5).

A RDL+Aspects também permite a construção de bibliotecas contendo padrões de reúso (*patterns*). Como por muitas vezes a construção dos pontos de extensão utiliza-se de mecanismos de reúso de projeto – como padrões de projeto [GAM95] para orientação a objetos e idiomas [HAN03] para orientação a aspectos – pelo fato destes mecanismos possuírem partes concretas e abstratas, as tarefas envolvidas no reúso destes pontos de extensão acabam sendo as mesmas. Os padrões de reúso procuram encapsular essas tarefas de reúso recorrentes em uma única estrutura que pode ser reutilizada em diversos *cookbooks*.

As principais características da RDL+Aspects são:

- **Declarações de controle de fluxo** – permitem controlar o fluxo de execução do programa, definindo atividades repetitivas e condicionais.
- **Declarações de variáveis** – permitem guardar valores originados por uma atividade de reúso.
- **Dependência de ordem de uma ação** – permite especificar uma dependência de ordem do tipo “atividade A vem antes de atividade B”.
- **Dependência de um estado futuro** – permite definir a obrigatoriedade ou não de um elemento no projeto final para que a atividade em questão possa ocorrer.

- **Declaração de atividades paralelas** – permite a execução de atividades de reúso disjuntas, podendo ser executadas em paralelo ou concorrentemente.
- **Comentários** – permite a introdução de texto em linguagem natural para auxiliar o entendimento do código.
- **Especificação dos pontos de extensão presentes em UML-AFR** – permite a especificação completa dos pontos de extensão definidos em UML-AFR.
- **Declaração de receitas rotuladas** – permite a definição de um conjunto de atividades afins em um único bloco identificado por um nome, que pode ser chamado diversas vezes de forma análoga a procedimentos de linguagens imperativas.

4.1.1 Estrutura da linguagem

As construções de mais alto nível da RDL+Aspects, assim como da RDL, são representadas por *cookbooks* (livro de receitas), *recipes* (receitas) e *patterns* (padrões). Um *cookbook* contém um conjunto de *recipes*. Uma *recipe* engloba uma série de tarefas de reúso relacionadas com uma determinada variabilidade. *Patterns* descrevem tarefas de reúso recorrentes encontradas durante o processo, como, por exemplo, o uso de padrões de projeto.

A estrutura de um programa de instanciação é mostrada no Quadro 6. Um programa de instanciação é definido por um *cookbook* de instanciação (**`instantiation cookbook`**) identificado por um nome, que deve ser igual ao nome do arquivo, seguido da declaração opcional[9] das bibliotecas de padrões de reúso utilizadas pelo *cookbook* (**`use <lista de bibliotecas>`**[10]). Esse *cookbook* deve obrigatoriamente conter uma *recipe* com o nome "**main**" contendo declarações (declarações de variáveis, atribuições, comandos e chamadas a rotinas) separadas por ponto-e-vírgula, que servirá de ponto de partida para a execução.

Outras *recipes* podem ser definidas a fim de melhor estruturar o código. A estrutura de um programa de composição (Quadro 7) é praticamente igual à de um programa de instanciação, mudando apenas a declaração do *cookbook*, que é definido como sendo de composição (**`composition cookbook`**). A declaração opcional **`requires instantiation`** é utilizada para especificar que a composição depende de dados determinados na instanciação, sendo necessária a execução desta etapa de reúso em primeiro lugar.

[9] Colchetes [ ] serão utilizados para identificar elementos opcionais.
[10] Os sinais < e > delimitam o espaço para elementos de acordo com a descrição informada.

```
instantiation cookbook MyCookBook [use <lista de bibliotecas>];
  recipe main();
    ...
  end_recipe;
  [<outras recipes>]
end_cookbook;
```

Quadro 6 – Estrutura básica de um programa de instanciação RDL+Aspects.
Fonte: O autor.

```
composition cookbook MyCookBook [requires instantiation]
                                [use <lista de bibliotecas>];
  recipe main();
    ...
  end_recipe;
  [<outras recipes>]
end_cookbook;
```

Quadro 7 – Estrutura básica de um programa de composição RDL+Aspects.
Fonte: O autor.

Ambos os programas (instanciação e composição) podem utilizar quaisquer comandos da linguagem RDL+Aspects, pois os elementos a serem modificados em cada etapa dependerão do projeto do FOA e das linguagens (POO e POA) utilizadas. Por exemplo, em AspectJ, para se definir um entrecorte de determinado adendo de um aspecto, é necessária a extensão do aspecto e a concretização do conjunto de junção associado ao adendo, informando os pontos da aplicação que serão interceptados; já com JBoss AOP [JBO03], as regras de composição são definidas em módulos independentes dos aspectos, permitindo o entrecorte sem que haja a extensão do aspecto. A restrição de comandos a serem usados em cada uma das etapas é sugerida para uma futura versão da RDL+Aspects, caso o uso freqüente da linguagem em situações práticas comprove esta necessidade.

```
recipe myRecipe1;
  x : number;
  c : class;
  x := 1;
  myRecipe2(x, c);
  /* x continua igual a 1
     c possui classe "Pessoa" */
end_recipe;

recipe myRecipe2(a : number, out z : class);
  // neste ponto, z é um ponteiro para c e a contém o valor 1
  z := new_class("Pessoa");
  a := a + 2;
  // a aqui é igual a 3
end_recipe;
```

Quadro 8 – Declaração de *recipes* RDL+Aspects.
Fonte: O autor.

A declaração de uma *recipe* é feita utilizando-se a palavra-chave **`recipe`** seguido do nome pelo qual ela será identificada, que deverá ser único (Quadro 8). Opcionalmente, poderá ser definida uma lista de parâmetros de entrada e/ou saída: parâmetros somente de entrada são passados por valor (o valor da variável externa não é modificado) e são declarados na forma **`<nome> : <tipo>`**; parâmetros de entrada e saída são passados por referência (o argumento deve ser uma variável externa e seu valor pode ser modificado internamente na *recipe*) e são declarados como **`out <nome> : <tipo>`**. A lista de parâmetros é separada por vírgulas. A RDL+Aspects não permite a declaração de estruturas que retornem valor (similares às funções das linguagens imperativas); as únicas estruturas capazes de retornar um valor são certos comandos da própria RDL+Aspects. A única maneira de se retornar valores de dentro de uma *recipe* é por parâmetros de entrada e saída. Deste modo, o comportamento de uma *recipe* é idêntico ao de um procedimento e não ao de uma função.

A chamada a uma *recipe* é feita pelo seu nome, passando-se uma lista de argumentos de acordo com os parâmetros definidos em sua declaração, como mostrado na linha 5 (Quadro 8). Caso a *recipe* não possua parâmetros, os parênteses podem ser omitidos tanto em sua declaração quanto em sua chamada. O corpo de uma *recipe* não pode conter a declaração de outra *recipe* (não existe aninhamento), mas as declarações de *recipes* podem ser feitas em qualquer ordem no código fonte, independentemente de possíveis dependências.

Comentários podem ser declarados pelas expressões **`//<comentário>`** (linha 11) – tudo o que segue até o final da linha será desconsiderado pelo compilador/interpretador – ou **`/*<comentário>*/`** (linhas 6-7), em que tudo que estiver entre os delimitadores será desconsiderado, inclusive quebras de linha.

```
1 pattern_library MyLibrary [use <lista de bibliotecas>];
2   pattern pattern1([<lista de parametros>]);
3     ...
4   end_pattern;
5   pattern pattern2([<lista de parametros>]);
6     ...
7   end_pattern;
8   ...
9 end_pattern_library;
```

Quadro 9 – Estrutura básica de uma biblioteca de padrões RDL+Aspects.
Fonte: O autor.

Os padrões de reúso (*patterns*) são organizados dentro de bibliotecas (*pattern libraries*). Uma biblioteca de padrões (Quadro 9) contém *patterns* relacionados, que funcionam praticamente da mesma forma que *recipes*, como se fossem rotinas, sendo a única diferença a possibilidade de *patterns* serem chamados de diferentes *cookbooks* (*recipes* somente podem

ser chamadas de dentro do *cookbook* que as define). Uma biblioteca de padrões de reúso RDL+Aspects é definida pela declaração **`pattern_library`**, seguida do nome da biblioteca – que também deverá ser igual ao nome do arquivo – e das demais bibliotecas utilizadas, contendo a declaração de um ou mais padrões (pela palavra-chave **`pattern`**).

4.1.2 Tipos de dados, variáveis e literais

A fim de manter a simplicidade da sintaxe e facilitar o desenvolvimento de um ambiente de execução, as versões mais antigas da RDL ([OLI01][OLI04]) eram fracamente tipadas, ou seja, suas variáveis e parâmetros não eram declarados como sendo de um único tipo; este era determinado implicitamente de acordo com o contexto. Os problemas da fraca tipagem de dados são as inconsistências e os comportamentos inesperados que podem surgir durante a execução dos programas, além de aumentar as chances de o programador inserir erros. Como a linguagem possui comandos que implicitamente retornam valores e recebem parâmetros de determinado tipo (p.ex., classes, métodos, atributos), não seria difícil de um programador não propositadamente atribuir um valor representando uma classe a uma variável e, logo após, passar esta variável como argumento para um parâmetro que espera um valor correspondente a um método ou atributo. Esses problemas se tornaram aparentes a partir do uso prático da linguagem [MEN05], sendo que a última versão da linguagem já é fortemente tipada, possuindo tipos de dados correspondentes àqueles encontrados em diagramas de classes UML e também tipos de dados comuns às linguagens de programação.

A RDL+Aspects estende e revisa o conjunto de tipos de dados da RDL, criando novos tipos para representar elementos relacionados a FOAs, além de definir explicitamente a declaração de variáveis e de parâmetros de *recipes* e *patterns* (já visto na Subseção 4.1.1).

4.1.2.1 Variáveis

Variáveis são posições de memória que armazenam valores durante a execução de um *cookbook*. Por ser fortemente tipada, todas as variáveis precisam ser declaradas antes de seu uso (atribuição ou leitura), embora não precisem ser declaradas em um lugar específico do

código. O uso de uma variável antes de sua declaração acarreta em um erro de compilação, quando o programa é executado na ferramenta apropriada.

Variáveis são declaradas na forma **<nome> : <tipo>** na qual *nome* é um identificador único e válido para a linguagem e *tipo* é obrigatoriamente um dos tipos definidos pela linguagem. Não é permitida a declaração de múltiplas variáveis (p.ex., **x, y, y : number**), nem a atribuição de valores na mesma declaração; todas as variáveis são iniciadas com um valor padrão de acordo com o seu tipo.

A nomeação de variáveis, *cookbooks*, *recipes*, *patterns* e parâmetros (identificadores) segue as mesmas regras de validade, que podem ser resumidas como a seguir:

- Nomes são seqüências ilimitadas de caracteres, que diferenciam maiúsculas de minúsculas e devem obrigatoriamente começar com uma letra (a-zA-Z);
- Os caracteres subseqüentes devem ser letras (a-zA-Z), dígitos (0-9) ou o caractere de *underscore* "_". Espaços e tabulações não são permitidos;
- Devem obrigatoriamente ser diferentes de qualquer palavra reservada ou palavra-chave da RDL+Aspects (comandos e declarações).

A atribuição de valores às variáveis foi modificada em relação à RDL, para evitar confusões em relação aos operadores introduzidos pela RDL+Aspects. A atribuição agora é feita pela construção **<nome_variável> := <expressão>**, na qual *nome_variável* representa o nome de uma variável previamente declarada e *expressão* corresponde à qualquer expressão que retorne um valor do mesmo tipo definido para a variável. Exemplos de declarações e atribuições de variáveis podem ser vistos no Quadro 10.

```
 1  x : number;     // declaração de variável "x" de tipo numérico
 2  x := 1;         // atribuição do valor numérico 1 a "x"
 3
 4  clazz : class;  // declaração de variável "clazz" de tipo classe
 5  clazz := new_class("Pessoa"); // atribuição do resultado da expressão
 6                                // à variavel "clazz"
 7
 8  a := a + 2;     // comando inválido, pois "a" ainda não foi declarado
 9  y, z : boolean; // declaração inválida, não pode ser múltipla
10  x : number;     // declaração inválida, já existe variável "x"
11
12  // nomes de variável inválidos
13  24horas : boolean;
14  _#pv : number;
15  method : method;
```

Quadro 10 – Declarações e atribuições de variáveis.
Fonte: O autor.

4.1.2.2 Tipos de dados

A RDL+Aspects possui tipos de dados que representam os elementos da UML envolvidos em diagramas de classes/aspectos pertinentes a FOAs, além de tipos adicionais para representar números, seqüências de caracteres, valores lógicos e vetores. Cada tipo possui um valor padrão de iniciação de variáveis. Também possui uma forma específica para a declaração de literais – valores expressos diretamente no código do *cookbook* – ou comandos para manipulação específicos da linguagem.

O Quadro 11 detalha as características de cada tipo da linguagem RDL+Aspects, mostrando sua descrição, o valor inicial de suas variáveis e a especificação de literais ou os comandos relacionados. Além desses tipos, existe um "tipo de dados" especial, utilizado internamente por parâmetros de alguns comandos: **`variant`**, que significa um valor de qualquer tipo. Esse tipo não pode ser usado em declarações de variáveis e parâmetros.

`string`	Seqüência de caracteres. Uma string é vazia quando não possui nenhum caractere; não existe valor nulo. Seus literais são caracteres delimitados por aspas duplas; caracteres aspas duplas ou contrabarra de uma string devem ser precedidos de contrabarra (\" e \\).	
	Valor inicial padrão	`""` – seqüência vazia
	Literais	`""  "a"  "hello \\world"  "com \"aspas\""`
`number`	Valor numérico, inteiro ou decimal.	
	Valor inicial padrão	`0` – zero
	Literais	`0  2  -179  3.14159  -2.54`
`boolean`	Valor lógico (verdadeiro ou falso).	
	Valor inicial padrão	`false` – valor falso
	Literais	`false  true`
`reusable`	Referência para um elemento reutilizável UML. É a classe topo da hierarquia de classes da RDL+Aspects, exatamente como apresentado na UML-AFR Figura 17. Todos os tipos UML (class, method, attribute, aspect, pointcut, advice, joinpoint) são suas subclasses diretas. Todas as referências RDL+Aspects podem ser nulas, ou seja, não referenciam nenhum objeto.	
	Valor inicial padrão	`null` – nulo
	Comandos associados	element_choice
`class`	Referência para um objeto representando uma classe UML	
	Valor inicial padrão	`null` – nulo
	Comandos associados	new_class, get_class, class_extension, select_class_extension, new_class_inheritance, inheritance_introduction
`method`	Referência para um objeto representando um método de classe UML.	
	Valor inicial padrão	`null` – nulo
	Comandos associados	new_method, get_method, method_extension, add_method_code, method_introduction
`attribute`	Referência para um objeto representando um atributo de classe UML.	
	Valor inicial padrão	`null` – nulo
	Comandos associados	new_attribute, get_attribute, value_assignment, value_selection, attribute_introduction

aspect	Referência para um objeto representando um aspecto UML.	
	Valor inicial padrão	`null` – nulo
	Comandos associados	new_aspect, get_aspect, aspect_extension, select_aspect_extension, new_aspect_inheritance, inheritance_introduction, method_introduction, attribute_introduction
pointcut	Referência para um objeto representando um conjunto de junção de aspecto UML.	
	Valor inicial padrão	`null` – nulo
	Comandos associados	new_pointcut, get_pointcut, pointcut_extension, add_joinpoint
advice	Referência para um objeto representando um adendo de aspecto UML.	
	Valor inicial padrão	`null` – nulo
	Comandos associados	new_advice, get_advice, add_advice_code
joinpoint	Referência para um objeto representando um possível ponto de junção de um sistema.	
	Valor inicial padrão	`null` – nulo
	Comandos associados	add_joinpoint
<tipo>[]	Vetor expansível de elementos do tipo especificado. O tamanho do vetor aumenta conforme a necessidade, e pode ter seu tamanho lido e/ou alterado pela propriedade **length** (p.ex., se `xy` contém 3 elementos, `xy.length` retorna 3, e modificar o valor de `xy.length` para 1 faz com que os dois últimos elementos sejam excluídos). O acesso aos elementos é com índice, começando em 0 (p.ex., `xy[2]` representa o 3º elemento do vetor).	
	Valor inicial padrão	`[]` – vetor vazio (`length` = 0)
	Literais	`[<elemento1>, <elemento2>, ...]`

Quadro 11 – Conjunto de tipos de dados da RDL+Aspects.
Fonte: O autor.

4.1.3 Operadores

Por ser fortemente tipada e possuir comandos que necessitam da avaliação de expressões lógicas (iterações e condicionais), a RDL+Aspects introduz operadores para realizar operações aritméticas, relacionais e lógicas, além de operadores específicos para manipulação de *strings* (Quadro 12). Cada operador recebe um ou dois operandos de determinado tipo, que podem ser valores literais, variáveis ou expressões.

Para que as expressões sejam corretamente avaliadas, os operadores seguem uma ordem de precedência definida no Quadro 13. Operadores de maior precedência são avaliados antes dos operadores de menor precedência. Caso operadores de mesma precedência apareçam na mesma expressão, estes serão avaliados da esquerda para a direita. A ordem de avaliação dos operadores de uma expressão pode ser mudada pelo uso de parênteses, sendo então avaliadas primeiro as expressões contidas entre parênteses, dos mais internos aos mais externos (Quadro 14).

Aritméticos	*op1* + *op2*	Adição de *op1* e *op2*, no caso de operandos numéricos.
	op1 - *op2*	Subtração de dois operandos numéricos.
	op1 * *op2*	Multiplicação de dois operandos numéricos.
	op1 / *op2*	Divisão de dois operandos numéricos.
	op1 % *op2*	Resto da divisão de dois operandos numéricos.
Relacionais	*op1* = *op2*	Igualdade entre dois valores numéricos ou lógicos.
	op1 <> *op2*	Não-igualdade entre dois valores numéricos ou lógicos.
	op1 > *op2*	Verdadeiro se *op1* numérico maior que *op2* numérico.
	op1 >= *op2*	Verdadeiro se *op1* numérico maior ou igual que *op2* numérico.
	op1 < *op2*	Verdadeiro se *op1* numérico menor que *op2* numérico.
	op1 <= *op2*	Verdadeiro se *op1* numérico menor ou igual que *op2* numérico.
Lógicos	not *op*	Negação de um operando lógico.
	op1 and *op2*	Operação lógica E entre dois operandos lógicos.
	op1 or *op2*	Operação lógica OU entre dois operandos lógicos.
	op1 xor *op2*	Operação lógica OU-Exclusivo entre dois operandos lógicos.
String	*op1* + *op2*	Operador de concatenação entre *strings*. Gera uma nova *string* a partir da concatenação de seus operandos, caso *op1* ou *op2* seja do tipo *string*. Operandos numéricos, lógicos e de tipos UML são convertidos para uma representação *string* e concatenados.
	lcfirst *op*	Coloca o primeiro caractere da *string* em minúscula.
	ucfirst *op*	Coloca o primeiro caractere da *string* em maiúscula.

Quadro 12 – Operadores da RDL+Aspects.
Fonte: O autor.

Precedência ↑ **maior**	unário (negação e ops. *string*)	not lcfirst ucfirst
	multiplicativo	* / %
	aditivo	+ -
	relacional	< <= > >=
	igualdade	= <>
	E lógico	and
	OU-Exclusivo lógico	xor
↓ **menor**	OU lógico	or

Quadro 13 – Ordem de precedência entre operadores.
Fonte: O autor.

```
// * e / tem precedência          // and tem precedência
3 + 3 * 6 - 3 / 3 → 20            true or true and false → true

// mudando com ( )                // mudando com ( )
(3 + 3) * ((6 - 3) / 3) → 6       (true or true) and false → false
```

Quadro 14 – Alterando a ordem de precedência entre operadores com parênteses.
Fonte: O autor.

4.1.4 Comandos

Os comandos da linguagem RDL+Aspects capturam as atividades que devem ser realizadas durante o processo de reúso de FOAs, como a criação de classes e aspectos, redefinição de métodos e conjuntos de junção, relações de herança, etc. Além disso, possui comandos comuns às linguagens de programação, como iterações, condicionais e entrada de dados. O conjunto de comandos RDL+Aspects amplia o conjunto original da RDL, modificando os comandos originais (principalmente definindo parâmetros e retornos fortemente tipados), removendo alguns (como os comandos relacionados a padrões *pattern_class_extension* e *pattern_method_extension*, já que padrões de reúso agora são definidos e chamados da mesma forma que *recipes*), e acrescentado novos comandos específicos para as atividades de reúso de FOAs.

As subseções a seguir apresentam todos os comandos da RDL+Aspects organizados em grupos. Cada comando é exibido em um quadro contendo:

- **Comando** – especifica o nome do comando em linguagem natural;
- **Sintaxe** – especifica a sintaxe do comando a ser usada nos programas. O tipo do retorno (se houver) é exibido antes do comando, e nomes de parâmetros exibidos em *itálico*;
- **Descrição** – fornece uma descrição textual do objetivo do comando;
- **Código** – mostra um exemplo de uso do comando dentro de um *cookbook*;
- **Projeto / Representação gráfica** – ilustra como o projeto é modificado pela execução do comando, caso este o modifique; caso contrário exibe uma representação gráfica da execução do comando.

4.1.4.1 Atividades Básicas

São atividades simples de manipulação de elementos de projeto, como a criação de classes, aspectos, métodos, atributos, conjuntos de junção e adendos. Também há comandos para a definição de relação de herança entre classes e entre aspectos, adição de código a métodos e adendos, e acesso a elementos existentes no modelo pelo nome.

Embora a criação de elementos – classes, métodos, atributos, aspectos, conjuntos de junção e adendos – necessite de mais informações que simplesmente seu nome (como tipos de

retorno, parâmetros, visibilidade, etc.), nesta versão da RDL+Aspects os comandos de criação de elementos recebem apenas o nome como parâmetro. Os comandos foram definidos dessa forma para manter simples tanto a sintaxe da linguagem quanto a implementação do ambiente de execução, bem como conseguir um grau mais elevado de independência da linguagem de programação do framework. Deste modo, quem fica responsável por implementar a semântica do comando, preenchendo as informações complementares, é o executor do comando: o próprio reutilizador, no caso de execução manual; ou o ambiente de execução, provavelmente com a interação do reutilizador, no caso de execução automatizada.

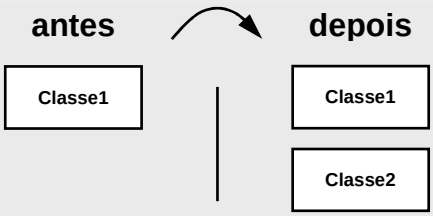
Os comandos de atividades básicas são apresentados abaixo (Quadro 15 a Quadro 30).

Comando	Criação de classe.
Sintaxe	`class` **`new_class`**`(`*`nome`* `: string)`
Descrição	Cria uma nova classe com o nome especificado. Retorna a classe criada.

Código

```
coobook Example;
  recipe main;
    c2 : class;
    // cria uma nova classe
    c2 := new_class("Classe2");
  end_recipe;
end_cookbook;
```

Projeto



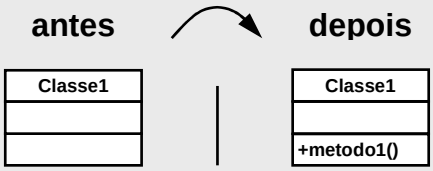
Quadro 15 – Comando de criação de classe.
Fonte: Oliveira [OLI07].

Comando	Criação de método.
Sintaxe	`method` **`new_method`**`(`*`classe`* `: class,` *`nome`* `: string)`
Descrição	Cria um novo método com o nome especificado e o adiciona na classe indicada. Retorna o método criado.

Código

```
coobook Example;
  recipe main;
    c1 : class;
    c1 := get_class("Classe1");
    // adiciona um novo método
    new_method(c1, "metodo1");
  end_recipe;
end_cookbook;
```

Projeto



Quadro 16 – Comando de criação de método.
Fonte: Oliveira [OLI07].

Comando Criação de atributo.

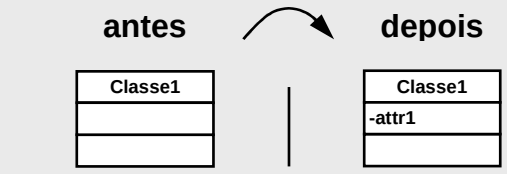
Sintaxe `attribute` **`new_attribute`**`(`*`classe`* `: class,` *`nome`* `: string)`

Descrição Cria um novo atributo com o nome especificado e o adiciona na classe indicada. Retorna o atributo criado.

Código

```
coobook Example;
  recipe main;
    c1 : class;
    c1 := get_class("Classe1");
    // adiciona um novo atributo
    new_attribute(c1, "attr1");
  end_recipe;
end_cookbook;
```

Projeto



Quadro 17 – Comando de criação de atributo.
Fonte: Oliveira [OLI07].

Comando Definição de herança entre classes.

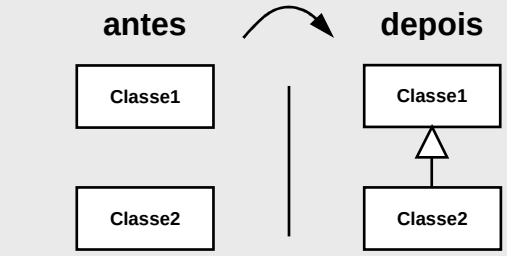
Sintaxe **`new_class_inheritance`**`(`*`super`* `: class,` *`sub`* `: class)`

Descrição Cria uma nova relação de herança entre a superclasse e a subclasse fornecidas.

Código

```
coobook Example;
  recipe main;
    c1 : class;
    c1 := get_class("Classe1");
    c2 : class;
    c2 := get_class("Classe2");
    // define herança
    new_class_inheritance(c1, c2);
  end_recipe;
end_cookbook;
```

Projeto



Quadro 18 – Comando de definição de herança entre classes.
Fonte: Oliveira [OLI07].

Comando Atribuição de código a um método.

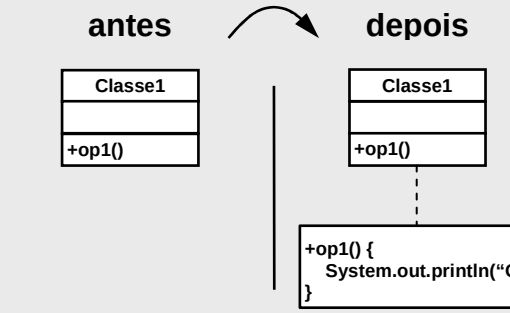
Sintaxe **`add_method_code`**`(`*`metodo`* `: method,` *`codigo`* `: string)`

Descrição Atribui o código fornecido a determinado método.
Visualmente, o código é exibido em uma nota UML anexada à classe.

Código

```
coobook Example;
  recipe main;
    c1 : class;
    c1 := get_class("Classe1");
    op1 : method;
    op1 := get_method(c1, "op1");
    // adiciona código ao método
    add_method_code(op1,
      "System.out.println(\"Ola\");"
    );
  end_recipe;
end_cookbook;
```

Projeto



Quadro 19 – Comando de atribuição de código a um método.
Fonte: Oliveira [OLI07].

Comando Obtenção de classe.

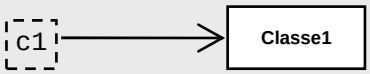
Sintaxe `class get_class(nome : string)`

Descrição Obtém uma referência à classe com o nome especificado.
Retorna a referência à classe ou nulo se não encontrada.

Código

```
coobook Example;
  recipe main;
    c1 : class;
    // obtém referência a Classe1
    c1 := get_class("Classe1");
  end_recipe;
end_cookbook;
```

Representação gráfica



Quadro 20 – Comando de obtenção de classe.
Fonte: O autor.

Comando Obtenção de método.

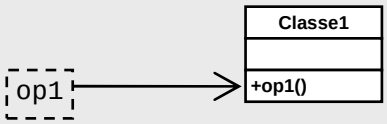
Sintaxe `method get_method(classe : class, nome : string)`

Descrição Obtém uma referência ao método da classe com o nome especificado.
Retorna a referência ao método ou nulo se não encontrado.

Código

```
coobook Example;
  recipe main;
    c1 : class;
    c1 := get_class("Classe1");
    op1 : method;
    // obtém referência a op1
    op1 := get_method(c1, "op1");
  end_recipe;
end_cookbook;
```

Representação gráfica



Quadro 21 – Comando de obtenção de método.
Fonte: O autor.

Comando Obtenção de atributo.

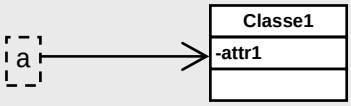
Sintaxe `attribute get_attribute(classe : class, nome : string)`

Descrição Obtém uma referência ao atributo da classe com o nome especificado.
Retorna a referência ao atributo ou nulo se não encontrado.

Código

```
coobook Example;
  recipe main;
    c1 : class;
    c1 := get_class("Classe1");
    a : attribute;
    // obtém referência a attr1
    a := get_attribute(c1, "attr1");
  end_recipe;
end_cookbook;
```

Representação gráfica



Quadro 22 – Comando de obtenção de atributo.
Fonte: O autor.

Comando Criação de aspecto.

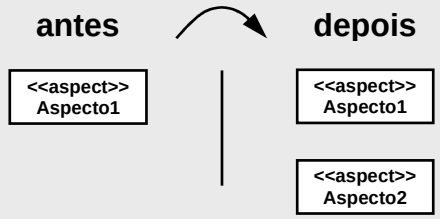
Sintaxe `aspect` **`new_aspect`**`(`*`nome`* `: string)`

Descrição Cria um novo aspecto com o nome especificado.
Retorna o aspecto criado.

Código

```
coobook Example;
  recipe main;
    a2 : aspect;
    // cria um novo aspecto
    a2 := new_aspect("Aspecto2");
  end_recipe;
end_cookbook;
```

Projeto



Quadro 23 – Comando de criação de aspecto.
Fonte: O autor.

Comando Definição de herança entre aspectos.

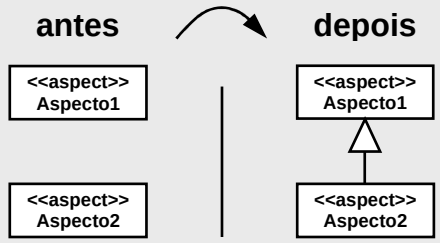
Sintaxe **`new_aspect_inheritance`**`(`*`super`* `: aspect,` *`sub`* `: aspect)`

Descrição Cria uma nova relação de herança entre o superaspecto e o subaspecto passados.

Código

```
coobook Example;
  recipe main;
    a1 : aspect;
    a1 := get_aspect("Aspecto1");
    a2 : aspect;
    a2 := get_aspect("Aspecto2");
    // define herança
    new_aspect_inheritance(a1, a2);
  end_recipe;
end_cookbook;
```

Projeto



Quadro 24 – Comando de definição de herança entre aspectos.
Fonte: O autor.

Comando Criação de conjunto de junção.

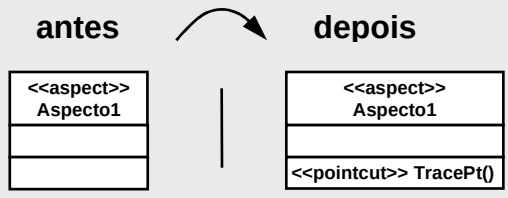
Sintaxe `pointcut` **`new_pointcut`**`(`*`aspecto`* `: aspect,` *`nome`* `: string)`

Descrição Cria um novo conjunto de junção com o nome especificado e o adiciona no aspecto indicado.
Retorna o conjunto de junção criado.

Código

```
coobook Example;
  recipe main;
    a1 : aspect;
    a1 := get_aspect("Aspecto1");
    // adiciona um novo pointcut
    new_pointcut(a1, "TracePt");
  end_recipe;
end_cookbook;
```

Projeto



Quadro 25 – Comando de criação de conjunto de junção.
Fonte: O autor.

Comando Criação de adendo.

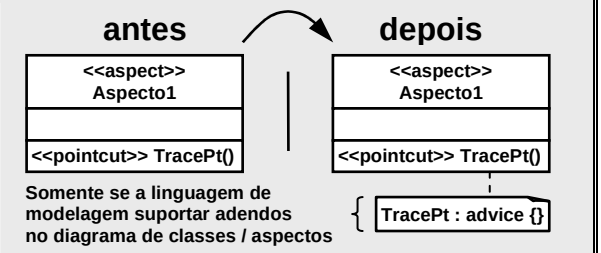
Sintaxe `advice new_advice(cJuncao : pointcut)`

Descrição Cria um novo adendo associado ao conjunto de junção.
Retorna o adendo criado.
A principio, o adendo não é uma construção visual pois não possui identificador e serve apenas para conter código associado ao conjunto de junção (adicionado pelo comando *add_advice_code*).
Caso a linguagem de modelagem suporte visualmente adendos, então eles serão exibidos visualmente.

Código

```
coobook Example;
 recipe main;
  a1 : aspect;
  a1 := get_aspect("Aspecto1");
  pc : pointcut;
  pc := get_pointcut(a1, "TracePt");
  // adiciona um novo adendo
  new_advice(pc);
 end_recipe;
end_cookbook;
```

Projeto



Quadro 26 – Comando de criação de adendo.
Fonte: O autor.

Comando Atribuição de código a um adendo.

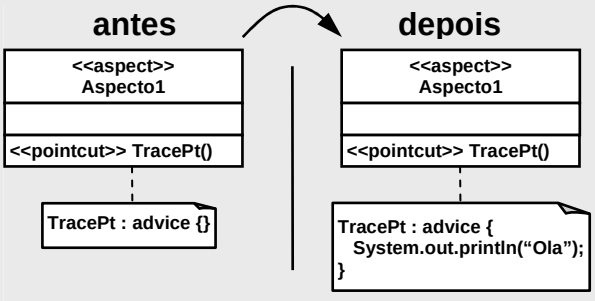
Sintaxe `add_advice_code(adendo : advice, codigo : string)`

Descrição Atribui o código fornecido a determinado adendo.
Visualmente, o código é exibido em uma nota UML anexada ao aspecto, caso a modelagem suporte visualização de adendos.

Código

```
coobook Example;
 recipe main;
  a1 : aspect;
  a1 := get_aspect("Aspecto1");
  pc : pointcut;
  pc := get_pointcut(a1, "TracePt");
  adendo : advice;
  adendo := get_advice(pc);
  // adiciona código ao adendo
  // associado ao pointcut TracePt
  add_advice_code(adendo,
    "System.out.println(\"Ola\");"
  );
 end_recipe;
end_cookbook;
```

Projeto



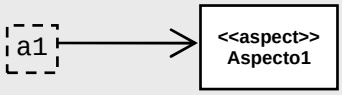
Quadro 27 – Comando de atribuição de código a um adendo.
Fonte: O autor.

Comando	Obtenção de aspecto.
Sintaxe	`aspect` **`get_aspect`**`(`*`nome`* `: string)`
Descrição	Obtém uma referência ao aspecto com o nome especificado. Retorna a referência ao aspecto ou nulo se não encontrado.

Código

```
coobook Example;
  recipe main;
    a1 : aspect;
    // obtém referência a Aspecto1
    a1 := get_aspect("Aspecto1");
  end_recipe;
end_cookbook;
```

Representação gráfica



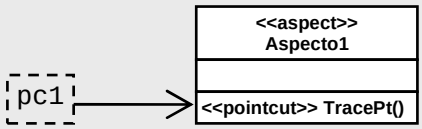
Quadro 28 – Comando de obtenção de aspecto.
Fonte: O autor.

Comando	Obtenção de conjunto de junção.
Sintaxe	`pointcut` **`get_pointcut`**`(`*`aspecto`* `: aspect,` *`nome`* `: string)`
Descrição	Obtém uma referência ao conjunto de junção do aspecto com o nome dado. Retorna a referência ao conjunto de junção ou nulo se não encontrado.

Código

```
coobook Example;
  recipe main;
    a1 : aspect;
    a1 := get_aspect("Aspecto1");
    pc : pointcut;
    // obtém referência a TracePt
    pc := get_pointcut(a1, "TracePt");
  end_recipe;
end_cookbook;
```

Representação gráfica



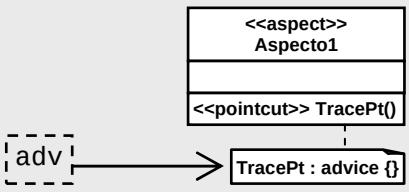
Quadro 29 – Comando de obtenção de conjunto de junção.
Fonte: O autor.

Comando	Obtenção de adendo.
Sintaxe	`advice` **`get_advice`**`(`*`cJuncao`* `: pointcut)`
Descrição	Obtém uma referência ao adendo associado ao conjunto de junção informado. Caso o conjunto de junção possua mais de um adendo, cabe ao reutilizador escolher qual será retornado Retorna a referência ao adendo ou nulo se não encontrado.

Código

```
coobook Example;
  recipe main;
    a1 : aspect;
    a1 := get_aspect("Aspecto1");
    pc : pointcut;
    pc := get_pointcut(a1, "TracePt");
    adv : advice;
    // obtém referência ao adendo
    adv := get_advice(pc);
  end_recipe;
end_cookbook;
```

Representação gráfica



Quadro 30 – Comando de obtenção de adendo.
Fonte: O autor.

4.1.4.2 Atividades de Reúso

São comandos que agrupam atividades básicas de forma a executar uma atividade específica de reúso para um elemento, como a especialização de classes e aspectos ou a redefinição de métodos e conjuntos de junção.

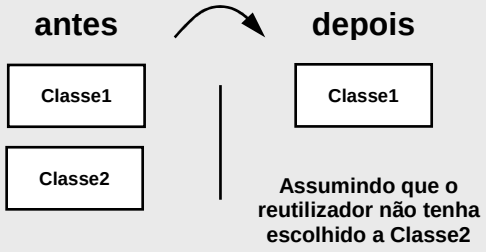
Os comandos de atividades de reúso são apresentados abaixo (Quadro 31 a Quadro 43).

Comando	Escolha de elemento.
Sintaxe	`boolean` **`element_choice`**`(`*`elemento`* `: reusable)`
Descrição	Pergunta ao reutilizador se o elemento informado estará presente ou não no projeto da aplicação final. Caso o elemento não seja escolhido, o mesmo é removido do projeto. Retorna verdadeiro se o elemento foi escolhido, e falso se não foi.

Código

```
coobook Example;
  recipe main;
    c2 : class;
    c2 := get_class("Classe2");
    // escolhe se a Classe2 fica
    element_choice(c2);
  end_recipe;
end_cookbook;
```

Projeto



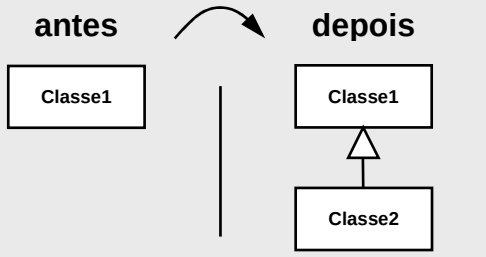
Quadro 31 – Comando de escolha de elemento.
Fonte: Oliveira [OLI07].

Comando	Extensão de classe.
Sintaxe	`class` **`class_extension`**`(`*`super`* `: class,` *`nome`* `: string)`
Descrição	Cria uma nova subclasse de *super* com o nome especificado. Retorna a subclasse criada. Esse comando utiliza os comandos de criação de classe e definição de herança, aumentando assim a expressividade da linguagem.

Código

```
coobook Example;
  recipe main;
    c1 : class;
    c1 := get_class("Classe1");
    // nova subclasse
    class_extension(c1, "Classe2");
  end_recipe;
end_cookbook;
```

Projeto



Quadro 32 – Comando de extensão de classe.
Fonte: Oliveira [OLI07].

Comando Extensão de classe por seleção.

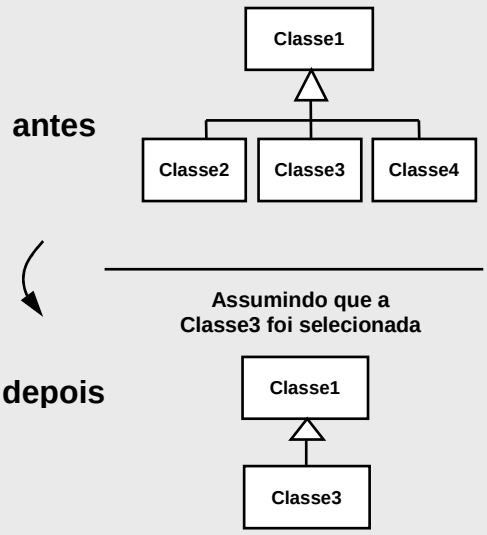
Sintaxe `class` **`select_class_extension`**`(`*`super`* `: class)`

Descrição Pede ao reutilizador para selecionar uma das subclasses concretas de *super*. Retorna a subclasse concreta selecionada.
Esse comando lista todas as subclasses concretas de *super*, permitindo assim a seleção de uma, sendo que as demais serão removidas do projeto final.

Código

```
coobook Example;
  recipe main;
    c1 : class;
    c1 := get_class("Classe1");
    // seleção de subclasse
    select_class_extension(c1);
  end_recipe;
end_cookbook;
```

Projeto



Quadro 33 – Comando de extensão de classe por seleção.
Fonte: Oliveira [OLI07].

Comando Redefinição de método.

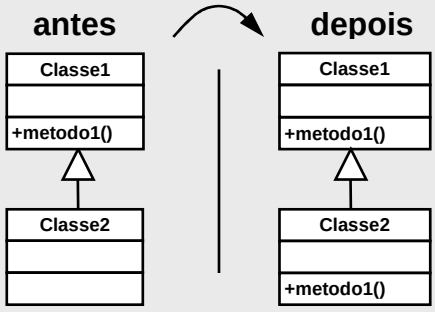
Sintaxe `method` **`method_extension`**`(`*`super`*`: class, `*`metodo`*`: method, `*`sub`*`: class)`

Descrição Redefine o método da superclasse na subclasse.
Retorna o novo método da subclasse.
Esse comando aumenta a expressividade da linguagem pois além da criação de um novo método verifica a relação de herança entre as classes.

Código

```
coobook Example;
  recipe main;
    c1 : class;
    c1 := get_class("Classe1");
    c2 : class;
    c2 := get_class("Classe2");
    m1 : method;
    m1 := get_method(c1, "metodo1");
    // redefine método na sub
    method_extension(c1, m1, c2);
  end_recipe;
end_cookbook;
```

Projeto



Quadro 34 – Comando de redefinição de método.
Fonte: Oliveira [OLI07].

Comando Atribuição de valor.

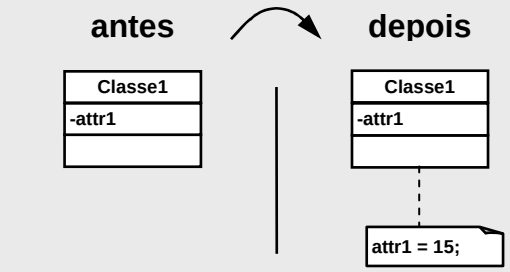
Sintaxe `value_assignment(`*atributo* `: attribute,` *valor* `: variant)`

Descrição Atribui o valor *val* ao atributo especificado.
A visualização da atribuição é feita por uma nota UML anexada à classe.

Código

```
coobook Example;
  recipe main;
    c1 : class;
    c1 := get_class("Classe1");
    a1 : attribute;
    a1 := get_attribute(c1, "attr1");
    // atribui valor ao atributo
    value_assignment(a1, 15);
  end_recipe;
end_cookbook;
```

Projeto



Quadro 35 – Comando de atribuição de valor.
Fonte: Oliveira [OLI07].

Comando Seleção de valor.

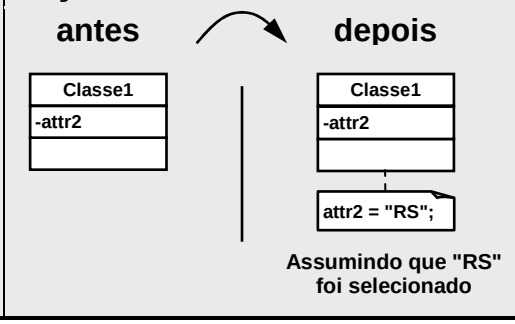
Sintaxe `value_selection(`*atributo* `: attribute,` *lista* `: variant[])`

Descrição Atribui o valor selecionado pelo reutilizador da lista informada ao atributo.
A visualização da atribuição é feita por uma nota UML anexada à classe.

Código

```
coobook Example;
  recipe main;
    c1 : class;
    c1 := get_class("Classe1");
    a2 : attribute;
    a2 := get_attribute(c1, "attr1");
    // atribui valor ao atributo
    value_selection(attr2, ["RS", "SC"]);
  end_recipe;
end_cookbook;
```

Projeto



Quadro 36 – Comando de seleção de valor.
Fonte: Oliveira [OLI07].

Comando Extensão de aspecto.

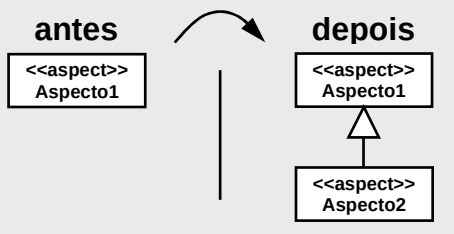
Sintaxe `aspect aspect_extension(`*super* `: aspect,` *nome* `: string)`

Descrição Cria um novo subaspecto de *super* com o nome especificado. Retorna o subaspecto criado. Esse comando utiliza os comandos de criação de aspecto e definição de herança, aumentando assim a expressividade da linguagem.

Código

```
coobook Example;
 recipe main;
   a1 : aspect;
   a1 := get_aspect("Aspecto1");
   // novo subaspecto
   aspect_extension(a1, "Aspecto2");
 end_recipe;
end_cookbook;
```

Projeto



Quadro 37 – Comando de extensão de aspecto.
Fonte: O autor.

Comando Extensão de aspecto por seleção.

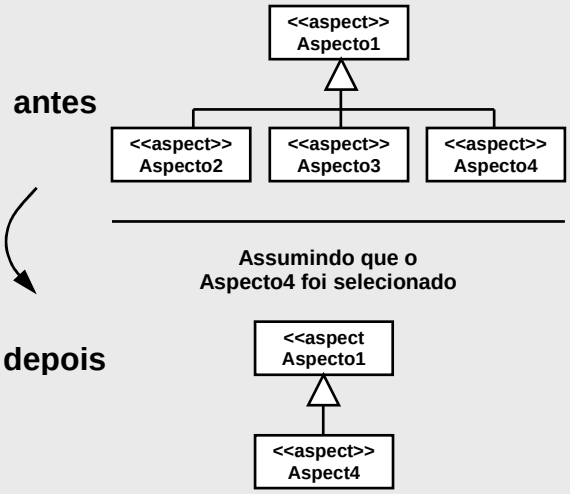
Sintaxe `aspect` **`select_aspect_extension`**`(`*`super`* `: aspect)`

Descrição Pede ao reutilizador para selecionar um dos subaspectos concretos de *super*. Retorna o subaspecto concreto selecionado.
Esse comando lista todos os subaspectos concretos de *super*, permitindo assim a seleção de um, sendo que os demais serão removidos do projeto final.

Código

```
coobook Example;
  recipe main;
    a1 : aspect;
    a1 := get_aspect("Aspecto1");
    // seleção de subaspecto
    select_aspect_extension(a1);
  end_recipe;
end_cookbook;
```

Projeto



Quadro 38 – Comando de extensão de aspecto por seleção.
Fonte: O autor.

Comando Redefinição de conjunto de junção.

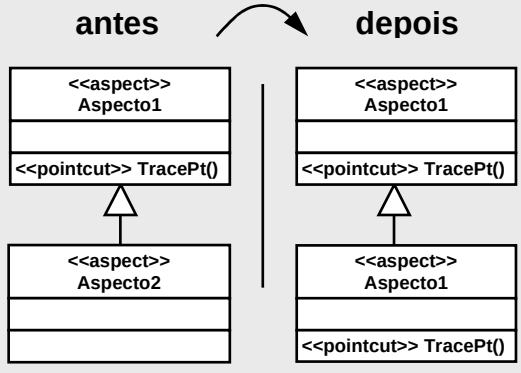
Sintaxe `pointcut` **`pointcut_extension`**`(`*`super`* `: aspect,` *`pc`* `: pointcut,`
`sub` `: aspect)`

Descrição Redefine o conjunto de junção especificado do superaspecto no subaspecto. Retorna o novo conjunto de junção do subaspecto.
Esse comando aumenta a expressividade da linguagem pois, além da criação de um novo conjunto de junção, verifica a relação de herança entre os aspectos.

Código

```
coobook Example;
 recipe main;
  a1 : aspect;
  a1 := get_aspect("Aspecto1");
  a2 : aspect;
  a2 := get_aspect("Aspecto2");
  pc : pointcut;
  pc := get_pointcut(a1, "TracePt");
  // redefine TracePt no sub
  pointcut_extension(a1, pc, a2);
 end_recipe;
end_cookbook;
```

Projeto



Quadro 39 – Comando de redefinição de conjunto de junção.
Fonte: O autor.

Comando Entrecorte (adição de pontos de junção).

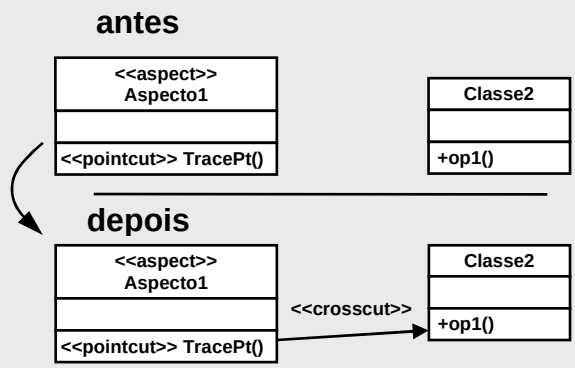
Sintaxe **add_joinpoint**(*cJuncao* : pointcut, *ptJuncao* : variant)

Descrição Adiciona o ponto de junção *ptJuncao* ao conjunto de junção especificado, ou seja, determina uma relação de entrecorte entre o aspecto e o ponto de junção. Os tipos de pontos de junção vão depender tanto da linguagem de POO quanto da linguagem de POA utilizada, sendo geralmente chamadas de métodos, mas podendo ser também classes inteiras, atributos, etc.
A visualização desse relacionamento de entrecorte depende da linguagem de modelagem orientada a aspectos utilizada. O ambiente de execução deve ser configurável para aceitar diversas formas de se representar esse relacionamento.

Código

```
coobook Example;
 recipe main;
  a1 : aspect;
  a1 := get_aspect("Aspecto1");
  pc : pointcut;
  pc := get_pointcut(a1, "TracePt");
  c2 : class;
  c2 := get_class("Classe2");
  // entrecorte
  add_joinpoint(pc,
            get_method(c2, "op1"));
 end_recipe;
end_cookbook;
```

Projeto



Quadro 40 – Comando de entrecorte.
Fonte: O autor.

Comando Introdução de herança entre classes.

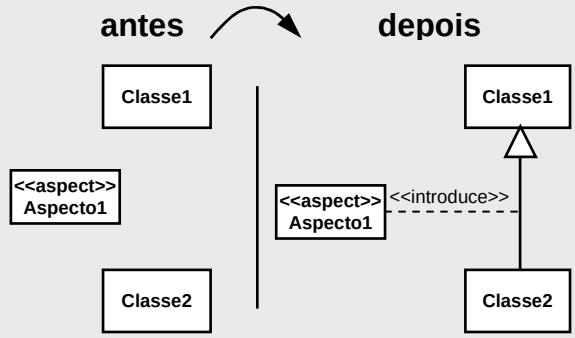
Sintaxe **inheritance_introduction**(*a* : aspect, *super* : class, *sub* : class)

Descrição Introduz uma nova relação de herança entre a superclasse e a subclasse via declaração intertipo feita pelo aspecto *a*.
Caso a linguagem de modelagem não possua mecanismos específicos para a representação da introdução da herança, ao menos a relação de herança deve ser representada.

Código

```
coobook Example;
  recipe main;
    a1 : aspect;
    a1 := get_aspect("Aspecto1");
    c1 : class;
    c1 := get_class("Classe1");
    c2 : class;
    c2 := get_class("Classe2");
    // define introdução de herança
    inheritance_introduction(a1,
                             c1, c2);
  end_recipe;
end_cookbook;
```

Projeto



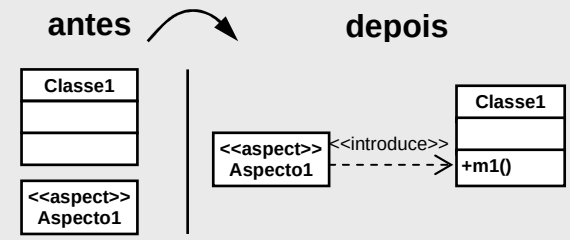
Quadro 41 – Comando de introdução de herança entre classes.
Fonte: O autor.

Comando	Introdução de método.
Sintaxe	`method` **`method_introduction`**`(`*`a`* `: aspect, `*`c`* `: class, `*`nome`* `: string)`
Descrição	Introduz um novo método com o nome especificado na classe via declaração intertipo feita pelo aspecto *a*. Retorna o método introduzido.

Código

```
coobook Example;
 recipe main;
  a1 : aspect;
  a1 := get_aspect("Aspecto1");
  c1 : class;
  c1 := get_class("Classe1");
  // introduz método
  method_introduction(a1, c1, "m1");
 end_recipe;
end_cookbook;
```

Projeto



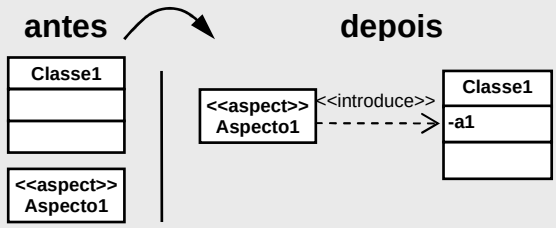
Quadro 42 – Comando de introdução de método.
Fonte: O autor.

Comando	Introdução de atributo.
Sintaxe	`attribute` **`attribute_introduction`**`(`*`a`*`: aspect, `*`c`*`: class, `*`nm`*`: string)`
Descrição	Introduz um novo atributo com o nome especificado na classe via declaração intertipo feita pelo aspecto *a*. Retorna o atributo criado.

Código

```
coobook Example;
 recipe main;
  a1 : aspect;
  a1 := get_aspect("Aspecto1");
  c1 : class;
  c1 := get_class("Classe1");
  // introduz atributo
  attribute_introduction(a1, c1, "a1");
 end_recipe;
end_cookbook;
```

Projeto



Quadro 43 – Comando de introdução de atributo.
Fonte: O autor.

4.1.4.3 Atividades de seqüência

Atividades de seqüência servem para descrever a forma como os comandos RDL+Aspects são combinados no fluxo de execução. Como algumas atividades de reúso podem depender de outras para a sua correta execução, o processo de reúso deve seguir a ordem estabelecida pelo desenvolvedor do framework. Além da relação de ordem implícita entre comandos, que é dada pela disposição dos comandos dentro do *cookbook*, como ocorre nas demais linguagens de programação, a RDL+Aspects oferece comandos que explicitam uma relação de ordem, seqüência ou paralelismo entre comandos. Também possui comandos

para expressar uma relação de dependência entre comandos e elementos de projeto. As atividades de seqüência possuem notação infixa para manter compatibilidade sintática com outras linguagens de programação, e serão apresentadas abaixo (Quadro 44 a Quadro 49).

Comando E (#).

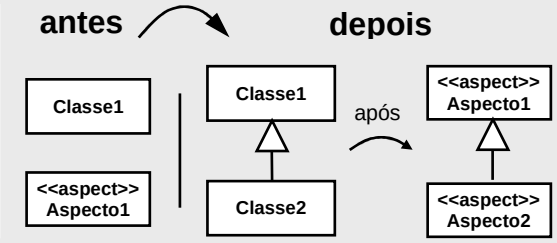
Sintaxe *cmd1* **#** *cmd2*

Descrição Define que ambos os comandos devem ser executados na ordem *cmd1*, *cmd2*.
Esse comando indica uma dependência de *cmd2* em relação a *cmd1*.
Essa relação de ordem é expressa implicitamente entre comandos quando estes são dispostos um após o outro no código, mas em alguns casos sua especificação explícita é necessária (quando os comandos devem estar na mesma linha ou para reforçar a idéia de dependência entre atividades importantes, por exemplo).

Código

```
coobook Example;
 recipe main;
  c1 : class;
  c1 := get_class("Classe1");
  a1 : aspect;
  a1 := get_aspect("Aspecto1");
  // seqüência E
  class_extension(c1, "Classe2") #
   aspect_extension(a1, "Aspecto2");
 end_recipe;
end_cookbook;
```

Projeto



Quadro 44 – Comando de seqüência E (#).
Fonte: Oliveira [OLI07].

Comando OU (o).

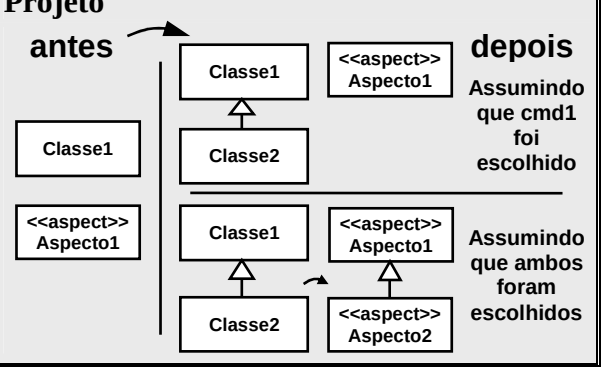
Sintaxe *cmd1* **o** *cmd2*

Descrição Interroga o reutilizador sobre qual comando, *cmd1* ou *cmd2*, será executado, ou ainda se ambos os comandos serão executados, na ordem.
Esse comando indica funcionalidades alternativas, mas que podem estar presentes ao mesmo tempo no projeto final.
Versões anteriores da RDL definiam esse comando como exclusivo, sendo que apenas um comando podia ser executado. Como isso fere a semântica da operação lógica OU, além da RDL fornecer outro comando que possui a lógica exclusiva, a semântica do comando OU foi alterada de modo a permitir a execução de um ou ambos os comandos.

Código

```
coobook Example;
 recipe main;
  c1 : class;
  c1 := get_class("Classe1");
  a1 : aspect;
  a1 := get_aspect("Aspecto1");
  // seqüência OU
  class_extension(c1, "Classe2") o
   aspect_extension(a1, "Aspecto2");
 end_recipe;
end_cookbook;
```

Projeto



Quadro 45 – Comando de seqüência OU (o).
Fonte: Oliveira [OLI07].

Comando OU-Exclusivo (xo).

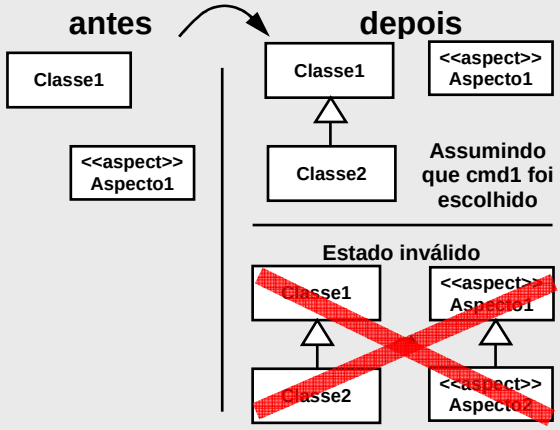
Sintaxe *cmd1* **xo** *cmd2*

Descrição Interroga o reutilizador sobre qual comando, *cmd1* ou *cmd2*, será executado. Os comandos são mutuamente exclusivos: ou *cmd1* ou *cmd2* será executado, mas não ambos.
Esse comando indica funcionalidades alternativas, mas que não podem estar presentes ao mesmo tempo no projeto final em razão de incompatibilidades funcionais.
Versões anteriores da RDL definiam esse comando como *requires exclusive*, sendo renomeado na RDL+Aspects para manter a coerência com as demais operações lógicas E e OU.

Código

```
coobook Example;
 recipe main;
  c1 : class;
  c1 := get_class("Classe1");
  a1 : aspect;
  a1 := get_aspect("Aspecto1");
  // seqüência OU-Exclusivo
  class_extension(c1, "Classe2") xo
   aspect_extension(a1, "Aspecto2");
 end_recipe;
end_cookbook;
```

Projeto



Quadro 46 – Comando de seqüência OU-Exclusivo (xo).
Fonte: O autor.

Comando Execução paralela.

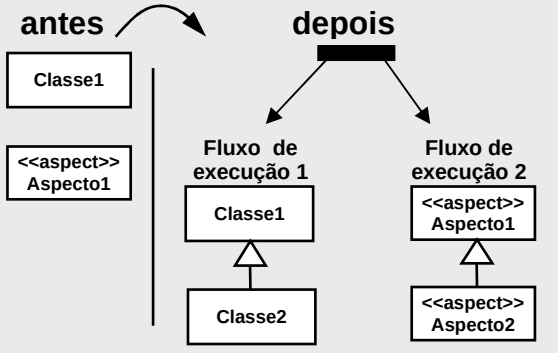
Sintaxe *cmd1* **||** *cmd2*

Descrição Define que ambos os comandos podem ser executados em qualquer ordem, paralela ou concorrentemente. Esse comando indica que não há dependências entre os comandos, podendo ser executados por desenvolvedores diferentes.
Vale salientar que a linguagem não possui mecanismos que previnem *deadlocks*.

Código

```
coobook Example;
 recipe main;
  c1 : class;
  c1 := get_class("Classe1");
  a1 : aspect;
  a1 := get_aspect("Aspecto1");
  // paralelo
  class_extension(c1, "Classe2") ||
   aspect_extension(a1, "Aspecto2");
 end_recipe;
end_cookbook;
```

Projeto



Quadro 47 – Comando de execução paralela.
Fonte: Oliveira [OLI07].

Comando Sincronização.

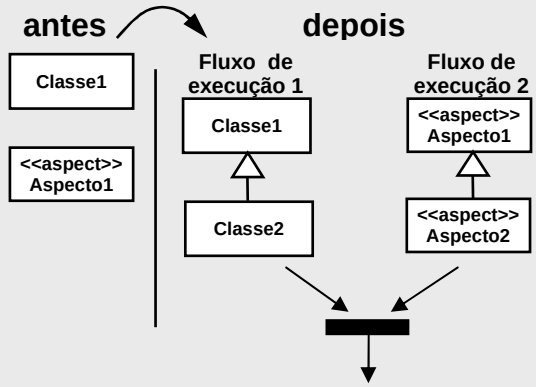
Sintaxe *cmd1* **sync** *cmd2*

Descrição Requer a sincronização dos dois comandos, ou seja, o processo de reúso não pode continuar até que ambos os comandos tenham sido executados.
Esse comando permite a sincronização de linhas de execução paralelas.

Código

```
coobook Example;
 recipe main;
  c1 : class;
  c1 := get_class("Classe1");
  a1 : aspect;
  a1 := get_aspect("Aspecto1");
  // sincronização
  class_extension(c1, "Classe2") sync
   aspect_extension(a1, "Aspecto2");
 end_recipe;
end_cookbook;
```

Projeto



Quadro 48 – Comando de sincronização.
Fonte: Oliveira [OLI07].

Comando Dependência de estado.

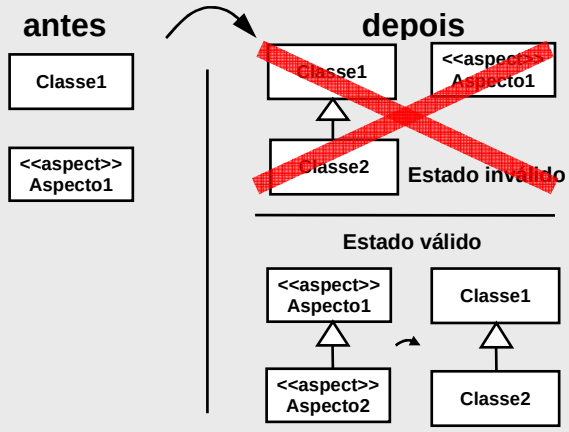
Sintaxe *cmd1* **requires** *reusable*

Descrição Requer a presença do elemento no projeto para a execução de *cmd1*.
Esse comando indica que *cmd1* depende do elemento para seu correto funcionamento. Caso o valor passado para *reusable* seja uma expressão (um comando que retorne um elemento), essa expressão será avaliada antes da execução de *cmd1*.

Código

```
coobook Example;
 recipe main;
  c1 : class;
  c1 := get_class("Classe1");
  a1 : aspect;
  a1 := get_aspect("Aspecto1");
  // dependência
  class_extension(c1, "Classe2")
      requires aspect_extension(a1,
                        "Aspecto2");
 end_recipe;
end_cookbook;
```

Projeto



Quadro 49 – Comando de dependência de estado.
Fonte: Oliveira [OLI07].

4.1.4.4 Comandos Diversos

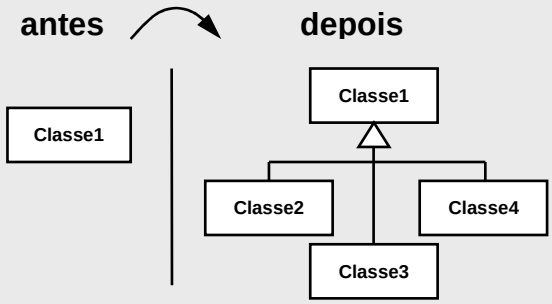
São comandos comuns às linguagens de programação, como declaração de iteração, declaração condicional e entrada de dados do usuário, necessários para a completitude da linguagem RDL+Aspects (Quadro 50 a Quadro 53).

Comando	Repetição (enquanto verdadeiro).
Sintaxe	**while** *boolean* **do** *cmd*; [*cmd*;]* **end_while;**
Descrição	Repete a execução de um ou mais comandos enquanto o valor da expressão lógica for verdadeira.

Código

```
coobook Example;
 recipe main;
  c1 : class;
  c1 := get_class("Classe1");
  i : number;
  i := 2;
  while i <= 4 do
   class_extension(c1, "Classe" + i);
   i := i + 1;
  end_while;
 end_recipe;
end_cookbook;
```

Projeto



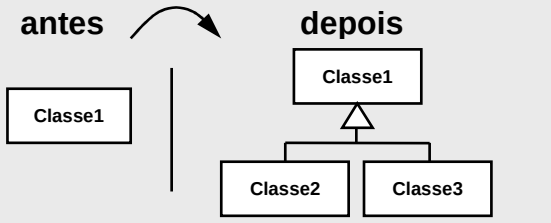
Quadro 50 – Comando de repetição.
Fonte: O autor.

Comando	Laço.
Sintaxe	**loop** *cmd*; [*cmd*;]* **end_loop;**
Descrição	Repete a execução de um ou mais comandos tantas vezes quantas o reutilizador desejar. Geralmente utilizada em conjunto com o comando de entrada de dados.

Código

```
coobook Example;
  recipe main;
    c1 : class;
    c1 := get_class("Classe1");
    loop
      class_extension(c1, ?);
    end_loop;
  end_recipe;
end_cookbook;
```

Projeto



Assumindo que o reutilizador executou 2 vezes e forneceu Classe2 e Classe3 como nomes de classe

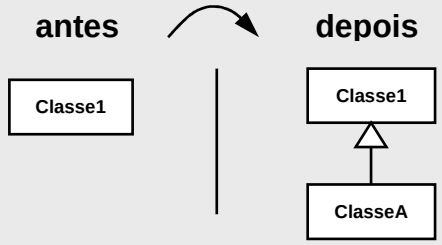
Quadro 51 – Comando de laço.
Fonte: Oliveira [OLI07].

Comando	Condicional (se).
Sintaxe	**if** *boolean* **then** *cmd*; [*cmd*;]* [**else** *cmd*; [*cmd*;]*] **end_if;**
Descrição	Executa um ou mais comandos de acordo com uma condição lógica. Se a condição for verdadeira, executa o(s) comando(s) após o **then**. A cláusula **else** é opcional: caso presente, seu(s) comando(s) serão executado se a condição for falsa.

Código

```
coobook Example;
 recipe main;
  c1 : class;
  c1 := get_class("Classe1");
  i : number;
  i := 2;
  if i > 4 then
   class_extension(c1, "Classe" + i);
  else
   class_extension(c1, "ClasseA");
  end_if;
 end_recipe;
end_cookbook;
```

Projeto



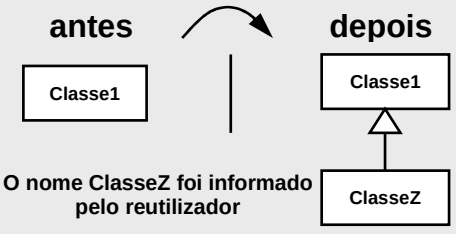
Quadro 52 – Comando condicional.
Fonte: O autor.

Comando	Interação com o reutilizador (entrada de dados).
Sintaxe	`variant ?`
Descrição	Entrada de dados: o reutilizador deve entrar com dados apropriados no ponto especificado. O tipo desses dados vai depender do contexto em que o comando está inserido. No caso de execução automatizada, o ambiente de execução deve interagir com o reutilizador perguntando quais os dados a serem inseridos no ponto indicado. Em última análise, esse é o principal comando responsável por incluir os requisitos específicos da aplicação final para dentro do processo de reúso.

Código

```
coobook Example;
  recipe main;
    c1 : class;
    c1 := get_class("Classe1");
    class_extension(c1, ?);
  end_recipe;
end_cookbook;
```

Projeto



Quadro 53 – Comando de interação com o reutilizador.
Fonte: Oliveira [OLI07].

4.2 RESUMO

O processo de reúso de um framework orientado a aspectos pode ser visto como uma seqüência ordenada de tarefas a serem realizadas para que uma instância válida deste

framework seja criada e/ou composta com algum código-base. Como um programa é uma seqüência de instruções por definição, nada mais natural que representar o processo de reúso como programas, sendo necessário apenas uma linguagem de programação que capture as possíveis tarefas de reúso em comandos apropriados. Uma das tecnologias da abordagem AFR proposta neste trabalho é a linguagem RDL+Aspects, que possui exatamente o objetivo de representar tarefas de reúso de FOAs como comandos de programação, de maneira formal. Deste modo o processo de reúso pode ser representado por programas capazes de serem processados por um computador.

A RDL+Aspects abstrai o processo de reúso em construções de alto nível como *cookbooks* (livro de receitas), *recipes* (receitas) e *patterns* (padrões), permitindo a criação de programas de instanciação e composição e de bibliotecas de padrões de reúso. Além disso, possui um conjunto de comandos, operadores e tipos de dados que permitem representar as tarefas envolvidas nas etapas de instanciação e composição do processo de reúso de um FOA.

Por manipular elementos no nível de projeto, a RDL+Aspects é independente do domínio do framework ou da linguagem de programação do mesmo, além de permitir uma melhor especificação da aplicação final antes da implementação, aumentando a correspondência entre projeto e implementação. As principais vantagens do uso da RDL+Aspects em relação a abordagens em linguagem natural são: a eliminação de ambigüidades, característica da linguagem natural; e a possibilidade da execução automatizada do processo de reúso por um computador, assistindo o desenvolvedor da aplicação a corretamente instanciar/compor o framework em questão.

5 A FERRAMENTA REUSE TOOL

A ferramenta *Reuse Tool* oferece um ambiente de execução para os programas (*cookbooks*) escritos em RDL+Aspects, guiando o desenvolvedor de aplicação por todas as etapas do processo de reúso. Ela possui o mesmo objetivo da ferramenta xFIT (*Framework Instantiation Tool*) [OLI01][OLI04][MEN05], que fornece um ambiente de execução para a linguagem RDL.

De acordo com a etapa de reúso – instanciação ou composição – sendo executada, a *Reuse Tool* recebe um ou dois diagramas e um *cookbook* RDL+Aspects, e, em conjunto com as informações fornecidas pelo desenvolvedor da aplicação, gera outro modelo. Os diagramas de classes/aspectos são recebidos e produzidos no formato XMI (*XML Metadata Interchange*) [OMG06b]. A *Reuse Tool* também executa tarefas de verificação sobre os elementos de projeto para garantir que a estrutura do modelo é regular e correta, como, por exemplo, garantir que métodos abstratos estejam redefinidos em classes concretas.

A *Reuse Tool* foi projetada para ser independente da abordagem utilizada para modelagem de aspectos. Para isso, a ferramenta utiliza um arquivo de configuração que define como os aspectos e seus elementos (conjuntos de junção, adendos, entrecortes) são representados na modelagem. A única restrição é que a abordagem de modelagem orientada a aspectos deve ser capaz de ser convertida para o formato XMI.

Os passos necessários para reusar um FOA utilizando-se a *Reuse Tool* e a RDL+Aspects dependem de quais etapas de reúso o framework em questão possui e a ordem de execução dessas etapas. A Figura 19 ilustra a operação da *Reuse Tool* para um FOA com ambas as etapas de reúso, em que a composição depende da instanciação:

- O desenvolvedor do framework fornece um programa RDL+Aspects contendo as atividades de instanciação e outro programa RDL+Aspects contendo as atividades de composição, em conjunto com o diagrama de classes/aspectos do mesmo;
- O desenvolvedor da aplicação opera a *Reuse Tool*, alimentando-a com o programa RDL+Aspects de instanciação e o diagrama de classes/aspectos do framework. Quando necessário, o desenvolvedor da aplicação fornece as informações requisitadas pela ferramenta de acordo com os requisitos específicos da aplicação;
- Ao final da etapa de instanciação, a *Reuse Tool* executa tarefas de verificação e reporta os erros encontrados, caso exista algum. Se nenhum erro for encontrado, o diagrama de classes/aspectos da instância do framework correspondente é gerado;

- O desenvolvedor da aplicação opera novamente a *Reuse Tool*, agora fornecendo o diagrama da instância do framework gerado no passo anterior, o programa RDL+Aspects de composição e o diagrama de classes/aspectos do código base com o qual o FT deve ser composto. Quando necessário, o desenvolvedor da aplicação novamente fornece informações de acordo com os requisitos específicos da aplicação; Ao final da etapa de composição a *Reuse Tool* executa as demais tarefas de verificação, reportando os erros encontrados. Se nenhum erro for encontrado, o diagrama de classes/aspectos da aplicação final (instância do framework composta com o código base) é gerado.
- Por fim, o desenvolvedor da aplicação pode utilizar uma ferramenta CASE para abrir o diagrama final e gerar os esqueletos para as classes e aspectos produzidos. Ao preencher esses esqueletos com o código apropriado o processo de reúso estará terminado.

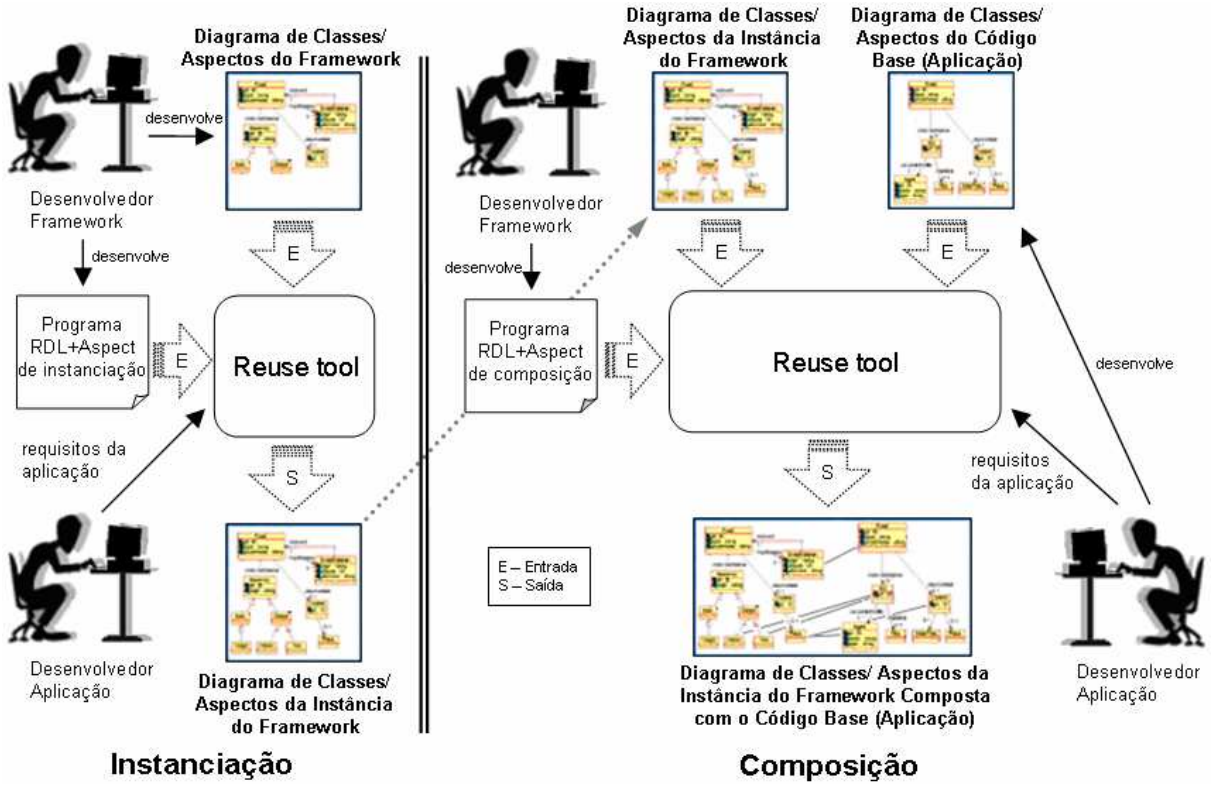


Figura 19 – Visão geral da operação da *Reuse Tool*.
Fonte: O autor.

Para as outras formas de reúso, a operação da *Reuse Tool* e da RDL+Aspects ocorre de maneira similar. Para FOAs com somente instanciação, o desenvolvedor do framework não precisa fornecer um programa de composição, e o diagrama produzido ao final da etapa de instanciação será o modelo da aplicação final. Para FOAs com somente composição, o

desenvolvedor do framework não precisa fornecer um programa de instanciação, e o diagrama do framework deve ser utilizado como entrada no lugar do diagrama da instância do framework. Para FOAs sem ordem específica entre as etapas de reúso, o diagrama de saída de uma etapa deve ser utilizado como entrada na outra etapa.

É importante ressaltar que ambos os *cookbooks* de instanciação e composição precisam representar as atividades de reúso de todos os pontos de extensão do framework para que uma aplicação seja gerada; caso os *cookbooks* representem parcialmente essas atividades, o reúso do framework produzirá outro artefato reutilizável, porém mais especializado [OLI01].

5.1 ARQUITETURA

A arquitetura da *Reuse Tool* é modularizada a fim de facilitar o seu desenvolvimento e manutenção, além de separar as responsabilidades envolvidas no processo assistido de reúso. Como mostrado na Figura 20, seus módulos interagem de modo a propiciar a compilação e a execução dos *cookbooks*. Essa execução modifica modelos representados em memória, que são importados de/exportados para XMI.

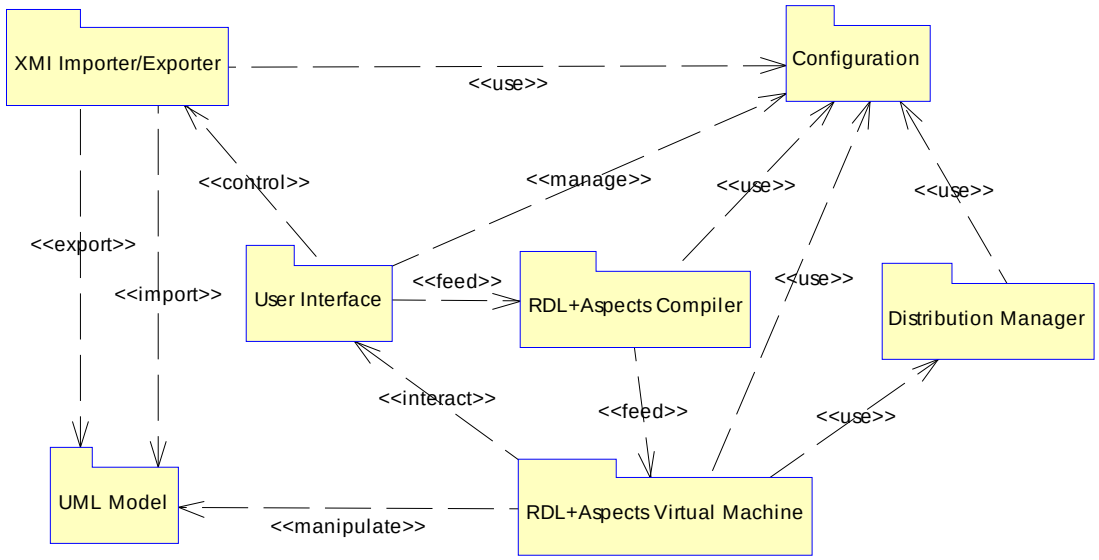


Figura 20 – Arquitetura da *Reuse Tool*.
Fonte: O autor.

Esses módulos são explicados em maiores detalhes abaixo:

- **Configuration** – responsável por manter as configurações da ferramenta em um arquivo. Com esse módulo é possível configurar: a linguagem de modelagem orientada a aspectos (como aspectos, conjuntos de junção, adendos e relações de

entrecorte são representados em XMI); as linguagens de programação do FOA (quais tipos de dados, quais propriedades – abstrato, estático, visibilidade, etc. – de classes, métodos, atributos, aspectos e conjuntos de junção); o local (diretório/caminho de rede) do repositório de bibliotecas de padrões de reúso; e informações para distribuição de tarefas paralelas/concorrentes (cliente/servidor, endereço de rede, porta).

- **UML Model** – responsável por manter os diagramas de classes em memória, permitindo sua manipulação pela máquina virtual (RDL+Aspects *Virtual Machine*) a medida que os *cookbooks* são executados. Esse módulo possui classes para representar a parte do metamodelo da UML referente aos diagramas de classes, com adição de classes para representar aspectos e seus elementos.
- **XMI Importer/Exporter** – módulo que realiza a conversão dos diagramas de classes/aspectos representados em XMI para um modelo em memória e vice-versa. Para garantir a flexibilidade da abordagem de modelagem orientada a aspectos, esse módulo utiliza informações da configuração da ferramenta para saber como as construções da orientação a aspectos são representadas nos arquivos XMI.
- **RDL+Aspects Compiler** – é o compilador da linguagem RDL+Aspects. É responsável por realizar a verificação sintática do *cookbook* e gerar o código executável para a máquina virtual RDL+Aspects.
- **RDL+Aspects Virtual Machine** – é a máquina virtual RDL+Aspects. Desempenha o papel de ambiente de execução para o código executável gerado pelo compilador, modificando o modelo em memória de acordo com os comandos definidos no *cookbook* e interagindo com o reutilizador através da interface gráfica. Também se comunica com o módulo de distribuição para realização de tarefas concorrentes/paralelas em diferentes instâncias da máquina virtual.
- **Distribution Manager** – módulo que permite a realização de tarefas paralelas ou concorrentes em diferentes máquinas físicas, realizando comunicação entre outras instâncias da *Reuse Tool* através da rede.
- **User Interface** – a interface gráfica da ferramenta que possibilita a interação do reutilizador, capturando seus gestos (digitação, movimentação e cliques do *mouse*) e exibindo as informações em telas amigáveis. Essas telas serão vistas mais adiante em maiores detalhes.

5.2 FUNCIONALIDADE

A *Reuse Tool* disponibiliza ao reutilizador três grandes funcionalidades, como pode ser observado na Figura 21: manter a configuração da ferramenta, carregar os artefatos necessários a uma etapa do processo de reúso e executar esta etapa.

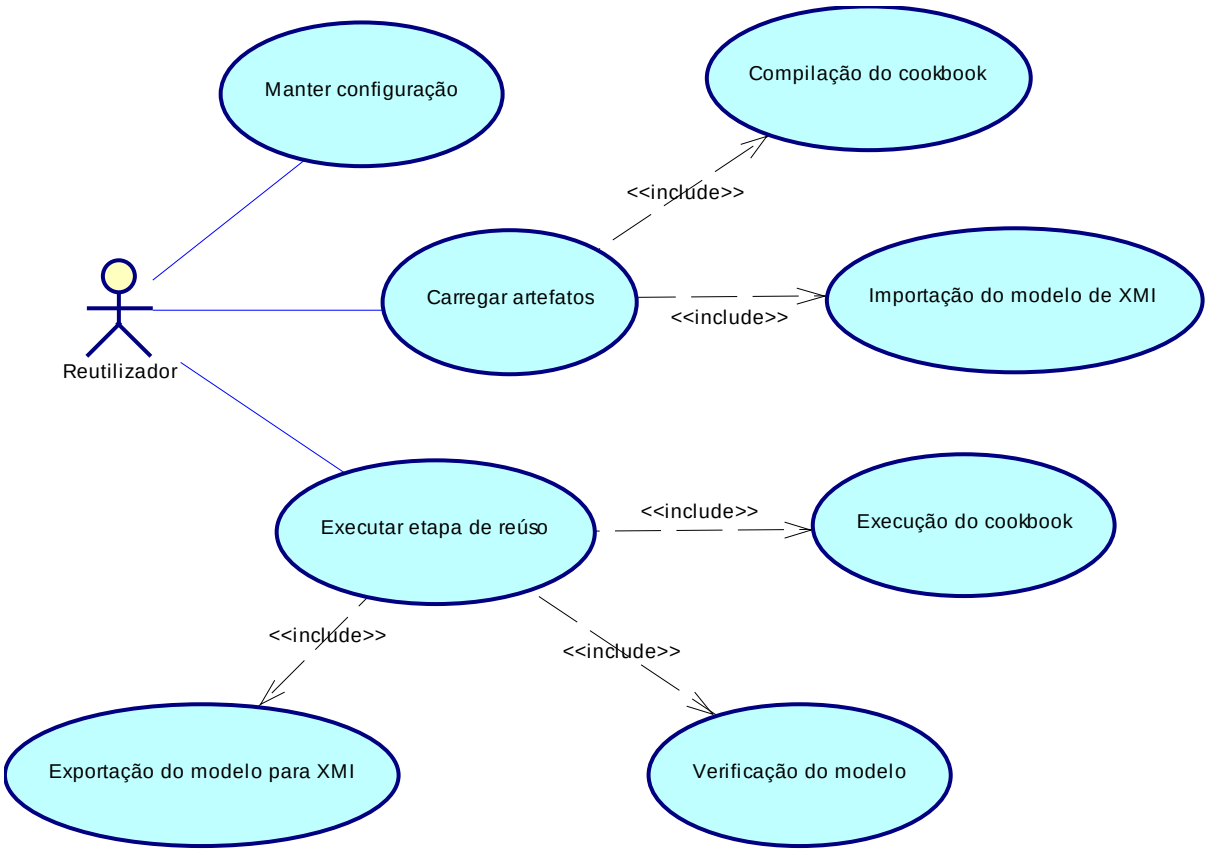


Figura 21 – Diagrama de casos de uso da *Reuse Tool*.
Fonte: O autor.

Ao manter a configuração da ferramenta, o reutilizador pode modificar tanto a linguagem de modelagem quanto as linguagens de programação do framework, bem como fornecer o local do repositório de bibliotecas de padrões de reúso. Também é possível ativar a distribuição de tarefas de reúso, configurando a ferramenta como servidor ou cliente e fornecendo o endereço e a porta de conexão.

A funcionalidade de carregar artefatos requisita os artefatos necessários para a etapa de reúso sendo executada – um diagrama de classes/aspectos e um *cookbook* quando instanciação e um ou dois diagramas de classes/aspectos (instância do framework e código-base juntos ou separados) e um *cookbook* quando composição. Internamente, essa funcionalidade verifica se o *cookbook* fornecido está de acordo com a etapa sendo realizada e

após o compila, e também realiza a importação dos diagramas em XMI para um modelo em memória.

Finalmente, a execução de uma etapa do processo de reúso começa com a execução propriamente dita do código gerado pela compilação do *cookbook*. Essa execução altera o modelo que está na memória de acordo com as atividades especificadas no *cookbook*. Após a execução, o modelo é verificado de forma a garantir as restrições impostas nos diagramas de entrada (verifica, por exemplo, se um determinado elemento marcado como obrigatório está presente no modelo final, se uma subclasse concreta implementa todos os métodos abstratos definidos na superclasse, etc.); caso alguma inconformidade seja encontrada, esta é reportada ao reutilizador, senão o modelo em memória é exportado para um arquivo XMI cujo destino é definido pelo reutilizador.

5.3 INTERFACE COM O USUÁRIO

Como o reúso de um framework consiste em adaptá-lo de acordo com os requisitos da aplicação sendo gerada, é muito importante a intervenção do reutilizador durante o processo: é com esta intervenção que o reutilizador fornece as informações pertinentes à aplicação sendo desenvolvida. Para que o reutilizador possa interagir facilmente com a ferramenta, sua interface foi desenvolvida de forma gráfica e amigável, em Java [SUN95] e baseada em janelas, a fim de integrar e completar a arquitetura anteriormente apresentada.

Assim que a ferramenta é executada, a tela principal ilustrada na Figura 22 é aberta, contendo um menu para acesso às funcionalidades disponíveis ao reutilizador:

- **File** – contém itens de menu para criar, abrir e salvar *cookbooks* RDL+Aspects, que são editados no quadro interno ***Cookbook Editor***, além de uma opção para sair da ferramenta.
- **Actions** – contém itens de menu para as funcionalidades disponíveis ao reutilizador, como configurar a ferramenta, carregar artefatos para uma etapa de reúso e executar a etapa de reúso previamente carregada.
- **Help** – contém um item de menu que abre o catálogo de ajuda ao reutilizador bem como um item para exibição de informações gerais sobre a ferramenta.

A tela principal inicialmente não possui nenhum quadro interno; estes são exibidos à medida que o reutilizador vai executando as funcionalidades. A ferramenta possui os seguintes quadros internos, conforme identificados na Figura 22:

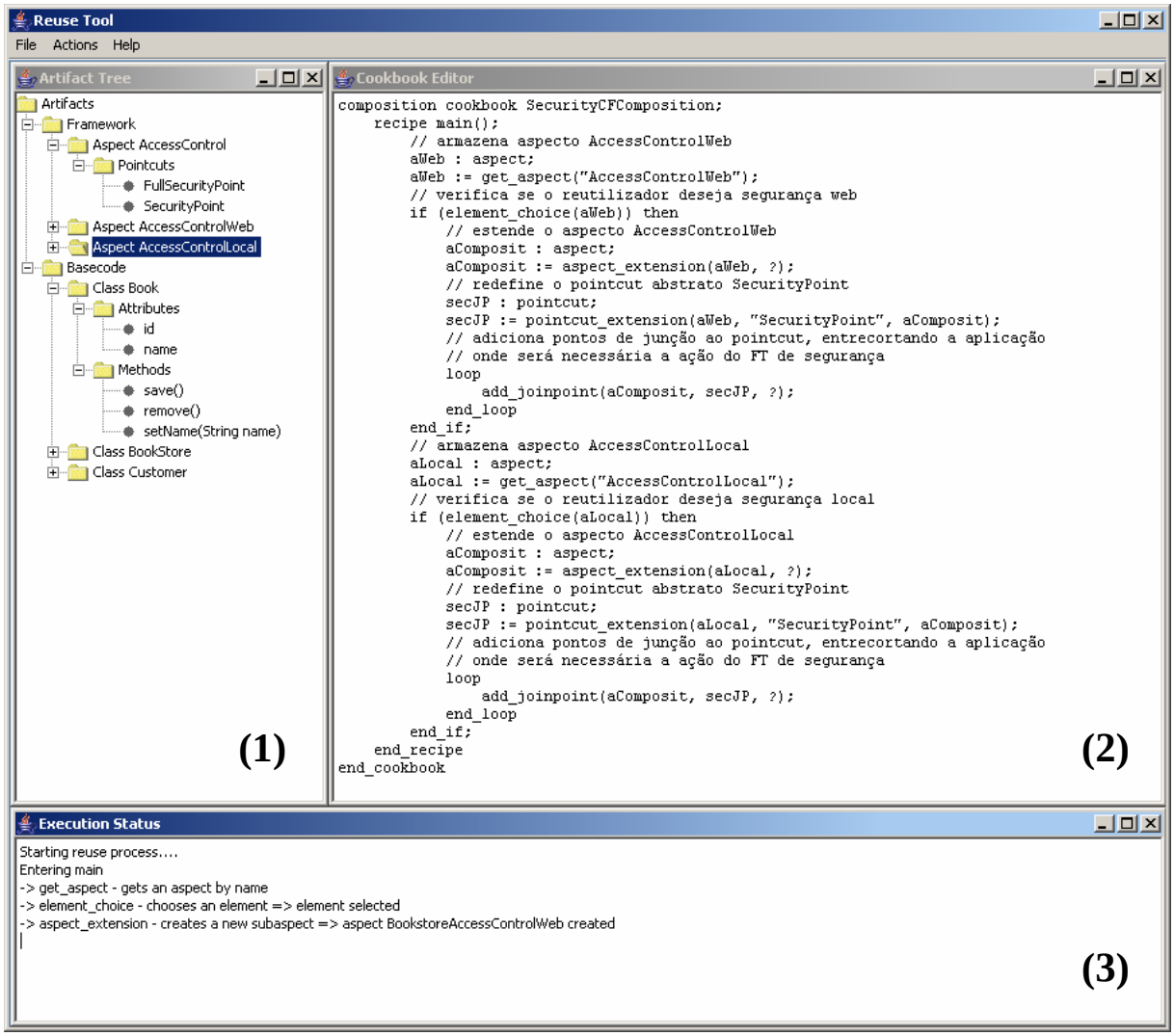


Figura 22 – Tela principal da *Reuse Tool*.
Fonte: O autor.

1. **Artifact Tree** – esse quadro é exibido após a carga dos artefatos envolvidos na etapa de reúso. Mostra todos os elementos – classes e aspectos e seus respectivos membros – contidos nos artefatos de forma hierárquica. Todas as alterações que ocorrem no modelo durante a etapa de reúso são refletidas na hierarquia, possibilitando ao reutilizador a rápida visualização das mesmas.
2. **Cookbook Editor** – é o editor de *cookbooks* RDL+Aspects, sendo exibido sempre que um *cookbook* é criado ou aberto (permitindo a edição), ou após a carga dos artefatos (bloqueando a edição).

3. **Execution Status** – exibe informações pertinentes à execução da etapa de reúso, servindo de orientação sobre o que já foi e o que está sendo feito para o reutilizador; também exibe erros encontrados na compilação de *cookbooks* ou na importação de arquivos XMI.

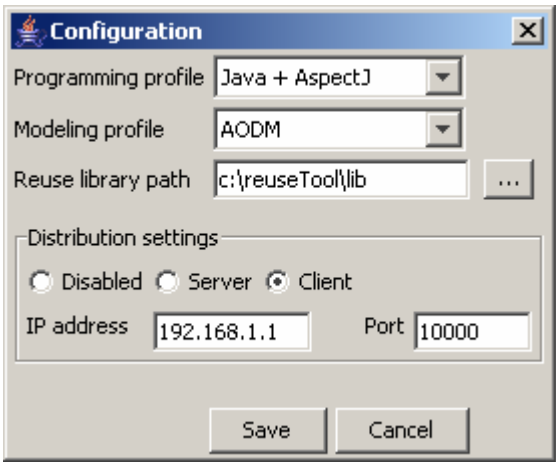


Figura 23 – Tela de configuração da ferramenta.
Fonte: O autor.

Além da tela principal, as funcionalidades também possuem telas auxiliares de forma a exibir e receber dados específicos. A tela de configuração (Figura 23) possibilita ao reutilizador informar: qual o perfil de programação do framework e do código-base (que define tipos de dados, propriedades de elementos, etc.); qual o perfil de modelagem (que define como os elementos da POA são representados em XMI); o caminho em que se encontram as bibliotecas de padrões de reúso; e as informações a respeito da distribuição de tarefas (para execução distribuída de tarefas paralelas/concorrentes).

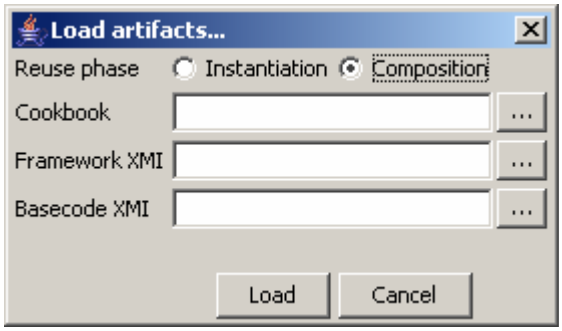


Figura 24 – Tela de carga de artefatos.
Fonte: O autor.

A funcionalidade de carregar artefatos também possui uma tela específica (Figura 24). Nessa tela é possível definir: qual etapa de reúso será realizada; o arquivo contendo o cookbook correspondente à etapa selecionada; o arquivo XMI contendo o diagrama de classes/aspectos do framework; e, caso seja selecionada a etapa de composição, o arquivo

XMI contendo o código-base ao qual o framework será acoplado. Após o reutilizador selecionar os artefatos e apertar o botão *Load* (Carregar), a ferramenta processa o *cookbook*, exibindo-o no quadro interno *Cookbook Editor* e compilando-o na seqüência. Também são processados os arquivos XMI, sendo estes importados para um modelo em memória, pronto para ser usado pela máquina virtual RDL+Aspects. Além disso, os elementos do modelo importado são exibidos de forma hierárquica no quadro *Artifact Tree*. Se existirem erros de compilação do *cookbook* ou de importação dos arquivos XMI, eles serão exibidos no quadro *Execution Status*.

Uma vez iniciada a execução da etapa de reúso, a *Reuse Tool* apresenta algumas telas relacionadas à semântica de alguns comandos RDL+Aspects, que necessitam de informações específicas do perfil de programação do framework e/ou da interação com o reutilizador. Os comandos que possuem telas específicas são: criação de classe, criação de método, criação de atributo, criação de aspecto, criação de conjunto de junção, criação de adendo, escolha de elemento, extensão de classe por seleção, seleção de valor, extensão de aspecto por seleção, seqüência OU, seqüência OU-Exclusivo, laço e interação com o reutilizador.

Todos os comandos de criação de elementos (classe, método, atributo, aspecto, conjunto de junção, adendo) e os de introdução de método e atributo possuem telas específicas para que o reutilizador forneça as informações adicionais necessárias de acordo com o perfil de programação configurado. Como exemplos, na Figura 25 são mostradas as telas dos comandos de criação de classe e de método para o perfil Java+AspectJ.

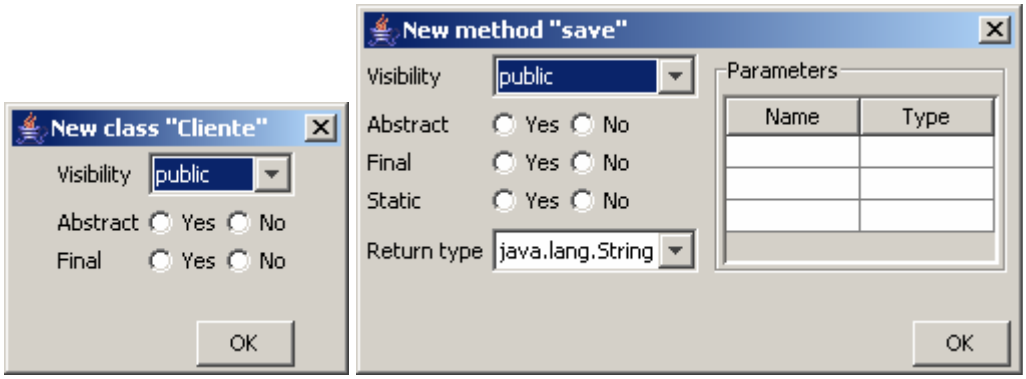


Figura 25 – Telas específicas dos comandos de criação de classe e de método.
Fonte: O autor.

O comando de escolha questiona o reutilizador se determinado elemento estará presente no projeto final ou se deve ser removido. Isso é feito com uma tela que exibe a pergunta contendo o nome do elemento e botões para Sim e Não (Figura 26).

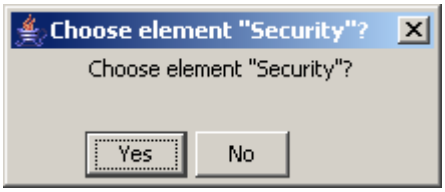


Figura 26 – Tela da escolha de elemento.
Fonte: O autor.

Os comandos de seleção apresentam uma lista de valores possíveis, dos quais o reutilizador deve selecionar um: para a extensão de classe por seleção são apresentadas as subclasses concretas da classe especificada; para a extensão de aspecto por seleção são apresentados os subaspectos concretos do aspecto especificado; e para a seleção de valor é apresentada a lista fornecida como parâmetro. Como exemplo é mostrada na Figura 27 a tela para o comando de extensão de classe por seleção.

Figura 27 – Tela da seleção de subclasse.
Fonte: O autor.

Os comandos de seqüência OU (o) e OU-Exclusivo (xo) também necessitam da interação com o reutilizador, que deve decidir qual comando será executado. Para isso, uma tela mostra quais as possíveis alternativas ao reutilizador, que seleciona as opções desejadas conforme as necessidades da aplicação sendo desenvolvida (Figura 28).

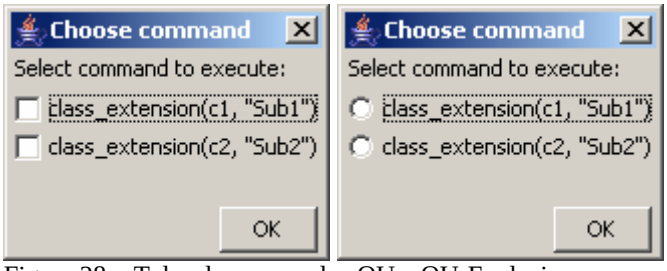


Figura 28 – Telas dos comandos OU e OU-Exclusivo.
Fonte: O autor.

O comando de laço repete a execução de um ou mais comandos tantas vezes quantas o reutilizador desejar. Para isso, após cada execução do bloco de comandos definido, a *Reuse Tool* questiona o reutilizador se deseja continuar repetindo ou não com a tela mostrada na Figura 29.

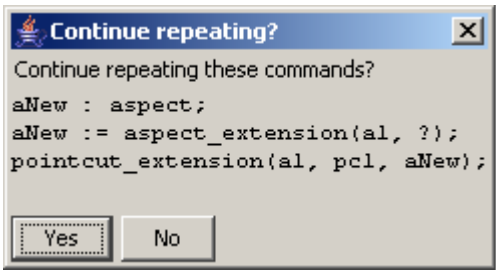


Figura 29 – Tela do comando de laço.
Fonte: O autor.

Por fim, o comando específico de interação também possui telas que possibilitam ao reutilizador entrar com informações em determinado ponto do *cookbook*. Como esse comando depende do contexto para retornar um valor, existem diversas telas, uma para cada contexto:

- **Contexto string ou numérico** – o comando abre uma tela contendo um campo onde o reutilizador poderá digitar o valor desejado (Figura 30). No caso do contexto numérico, só dígitos e o separador decimal são permitidos;

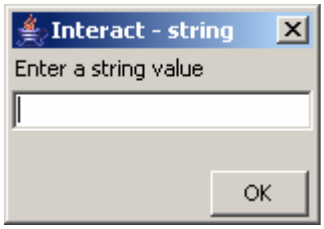


Figura 30 – Interação string.
Fonte: O autor.

- **Contexto booleano** – abre uma tela que possibilita a escolha ou do valor verdadeiro ou do falso;

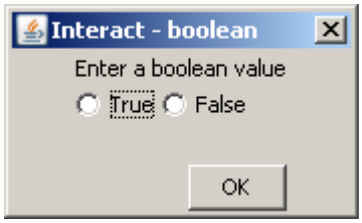


Figura 31 – Interação booleana.
Fonte: O autor.

- **Contexto de um elemento UML** – de acordo com o elemento em questão (classe, método, atributo, aspecto, conjunto de junção, adendo), abre uma tela específica onde o reutilizador deve informar o nome do elemento desejado. Após, executa internamente o comando de busca apropriado, passando o nome informado;

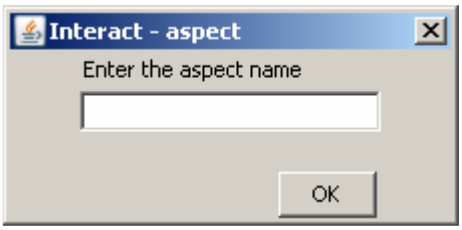


Figura 32 – Interação de elemento UML.
Fonte: O autor.

- **Contexto de ponto de junção** – abre uma tela em que possibilita fornecer o ponto de junção, pela escolha de um elemento do projeto (tipo e nome).

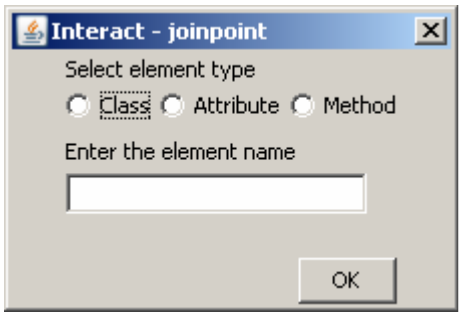


Figura 33 – Interação de ponto de junção.
Fonte: O autor.

5.4 CONSIDERAÇÕES FINAIS

Como o processo de reúso de um FOA pode se tornar uma tarefa árdua, o uso de uma ferramenta que auxilie o reutilizador diminui o esforço necessário para o desenvolvimento de uma aplicação. A ferramenta *Reuse Tool* foi desenvolvida com o objetivo de assistir o processo de reúso de FOAs.

A *Reuse Tool* facilita a execução do processo de reúso, aumentando o controle sobre o mesmo por parte do reutilizador. Também possibilita a verificação do modelo da aplicação final, diminuindo possíveis erros que a execução manual do reúso poderia causar.

6 EXEMPLOS DE USO DA AFR

Para ilustrar o completo funcionamento da abordagem AFR e suas tecnologias (UML-AFR, RDL+Aspects e *Reuse Tool*), foram feitas provas de conceito com alguns FOAs:

- o FT de segurança [CAM04a] já abordado no Capítulo 3;
- o FT de persistência apresentado em [CAM06];
- o FAOA utilizado pela empresa Embratec Good Card [EMB06].

Os pontos de extensão desses frameworks foram identificados nos diagramas de classes/aspectos de projeto utilizando-se a notação UML-AFR em conjunto com uma versão da abordagem AODM capaz de ser traduzida para XMI. Também foram codificados, em RDL+Aspects, os processos de reúso (instanciação e/ou composição) desses frameworks, a fim de verificar se os comandos da linguagem atendem às necessidades de cada processo. Com a utilização da ferramenta *Reuse Tool*, os *cookbooks* criados foram executados a fim de gerar uma nova aplicação e/ou de compô-los com uma aplicação já existente.

6.1 FRAMEWORK TRANSVERSAL DE SEGURANÇA

O FT de segurança mostrado em [CAM04a] é bem simples, contendo apenas a fase de composição em seu processo de reúso. Ele foi desenvolvido apenas para demonstrar algumas técnicas de projeto de variabilidades, de extensibilidade e de composição, contornando algumas restrições da linguagem AspectJ. A sua modelagem com UML-AFR em conjunto com AODM já foi exibida na Figura 18, sendo reproduzida abaixo apenas para facilitar a visualização por parte do leitor (Figura 34).

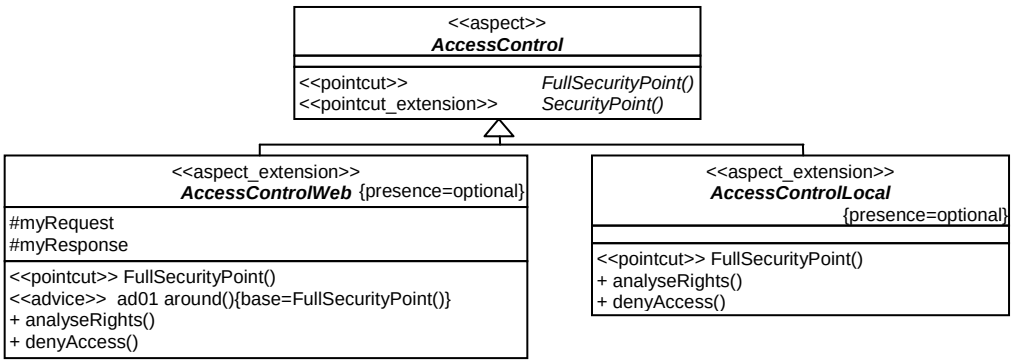


Figura 34 – FT de segurança modelado com UML-AFR e AODM (reprodução).
Fonte: O autor.

A etapa de composição desse framework consiste na especialização de um dos dois aspectos disponíveis (*AccessControlWeb* para aplicações *web* ou *AccessControlLocal* para aplicações locais). Após, no subaspecto criado, é feita a concretização do conjunto de junção SecurityPoint, informando quais pontos de junção da aplicação devem ser interceptados pelo framework. O *cookbook* de composição RDL+Aspects para esse framework está descrito no Quadro 54 abaixo.

```
 1 composition cookbook SecurityCFComposition;
 2
 3   recipe main();
 4     // armazena aspecto AccessControlWeb
 5     aWeb : aspect;
 6     aWeb := get_aspect("AccessControlWeb");
 7     // verifica se o reutilizador deseja segurança web
 8     if (element_choice(aWeb)) then
 9       // estende o aspecto AccessControlWeb
10       aComposit : aspect;
11       aComposit := aspect_extension(aWeb, ?);
12       // redefine o pointcut abstrato SecurityPoint
13       secJP : pointcut;
14       secJP := get_pointcut(aWeb, "SecurityPoint");
15       secJP := pointcut_extension(aWeb, secJP, aComposit);
16       // adiciona pontos de junção ao pointcut, entrecortando a
17       // aplicação, onde será necessária a ação do FT de segurança
18       loop
19         add_joinpoint(secJP, ?);
20       end_loop;
21     end_if;
22     // armazena aspecto AccessControlLocal
23     aLocal : aspect;
24     aLocal := get_aspect("AccessControlLocal");
25     // verifica se o reutilizador deseja segurança local
26     if (element_choice(aLocal)) then
27       // estende o aspecto AccessControlLocal
28       aComposit : aspect;
29       aComposit := aspect_extension(aLocal, ?);
30       // redefine o pointcut abstrato SecurityPoint
31       secJP : pointcut;
32       secJP := get_pointcut(aLocal, "SecurityPoint");
33       secJP := pointcut_extension(aLocal, secJP, aComposit);
34       // adiciona pontos de junção ao pointcut, entrecortando a
35       // aplicação, onde será necessária a ação do FT de segurança
36       loop
37         add_joinpoint(secJP, ?);
38       end_loop;
39     end_if;
40   end_recipe
41
42 end_cookbook
```

Quadro 54 – *Cookbook* de composição do FT de segurança.
Fonte: O autor.

Na linha 8 o reutilizador decide se o aspecto referente à segurança para aplicações *web* será utilizado ou não. Se sim, esse aspecto é especializado na linha 11, criando assim um

subaspecto com o nome definido pelo reutilizador, e redefinindo o conjunto de junção *SecurityPoint* dentro do subaspecto recém criado (linha 15). Na seqüência, o reutilizador informa quais os pontos de junção que irão compor o conjunto de junção redefinido, quantas vezes ele desejar, pelo comando de laço (linhas 18-20). As linhas 23 a 39 realizam as mesmas tarefas para o aspecto de segurança local. Desta forma o reutilizador fica responsável apenas por informar as características particulares da aplicação quando necessário.

Uma simples aplicação local de agenda telefônica foi utilizada para ilustrar a etapa de composição desse framework utilizando o *cookbook* acima. Essa aplicação contém apenas duas classes: *Agenda* e *Contato*, sendo uma agenda um container para contatos. Ambas as classes possuem operações de inclusão, edição, remoção e consulta, sendo que apenas as operações de consulta de ambas as classes não devem ter seu acesso protegido.

Para compor o FT a essa aplicação foi utilizada a ferramenta *Reuse Tool*, passando como artefatos o *cookbook* acima e os diagramas de classes/aspectos do framework e da aplicação no formato XMI. Ao executar o processo de reúso, o reutilizador não escolheu o aspecto *web* e sim o local, respondendo não à pergunta da linha 8 e sim a da linha 26. Após, definiu o nome do aspecto especializado sendo "AgendaAccessControl", e por último adicionou como pontos de junção os métodos de inclusão, edição e remoção das classes *Agenda* e *Contato*. Ao final do processo, a ferramenta gerou corretamente o projeto da aplicação final, que é mostrado na Figura 35 abaixo.

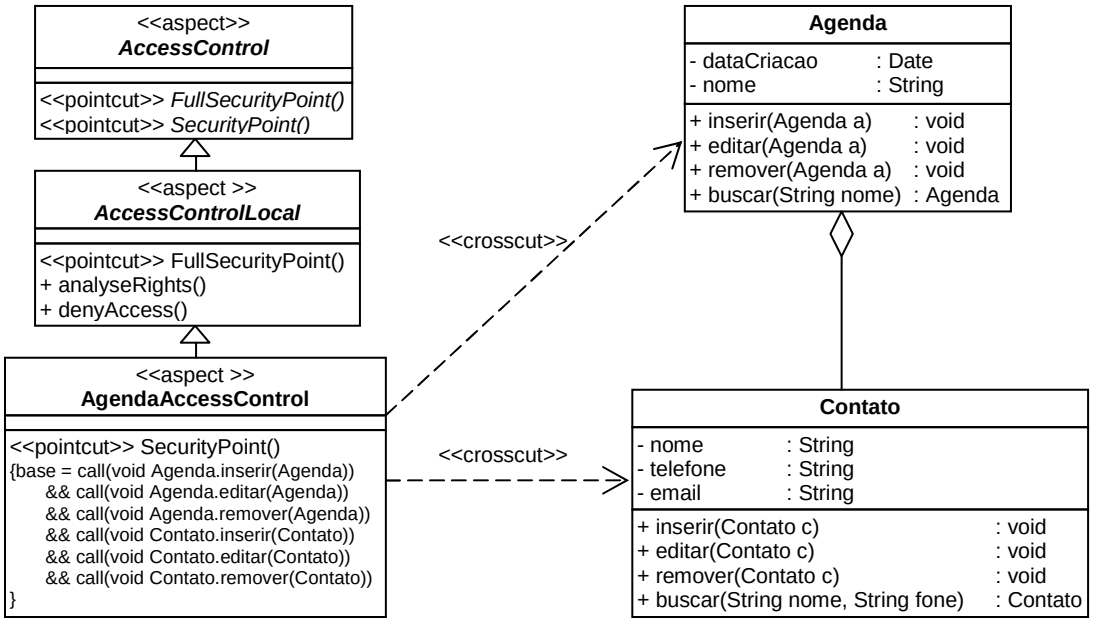


Figura 35 – Projeto final da aplicação de agenda composta com o FT de segurança.
Fonte: O autor.

6.2 FRAMEWORK TRANSVERSAL DE PERSISTÊNCIA

Em [CAM06] é apresentado um framework transversal de persistência que tem como objetivo facilitar o desenvolvimento de aplicações orientadas a objetos utilizando banco de dados relacionais. Os mecanismos necessários para o mapeamento de entidades relacionais para objetos e vice-versa são implementados pelo framework, podendo ser reutilizados durante o desenvolvimento de novas aplicações. Esse framework foi desenvolvido em AspectJ utilizando conceitos próprios desta linguagem como introduções, aspectos abstratos e redefinições de conjuntos de junção.

O FT consiste de dois módulos com objetivos distintos. O módulo de **operações persistentes** é um conjunto de operações relacionadas à persistência (inclusão, atualização, remoção e busca) que devem ser herdadas por classes de aplicação persistentes. A estratégia de implementação do módulo é introduzir essas operações persistentes em uma interface e fazer com que as classes de aplicação persistentes implementem essa interface. O módulo de **conexão**, do qual o módulo anterior é dependente, trata da conexão com o banco de dados. Esse módulo deve identificar pontos do código-base propícios para a abertura e fechamento da conexão com o banco de dados.

O módulo de operações persistentes foi implementado com as variabilidades de "Consciência Total" e "Consciência Parcial", permitindo assim que durante o reúso o desenvolvedor da aplicação escolha o que for mais adequado às suas necessidades. Quando a primeira estratégia é utilizada, o desenvolvedor é responsável por invocar todos os métodos de persistência disponibilizados pelo FT, enquanto que na segunda deve se preocupar apenas com as operações de remoção e busca.

O módulo de conexão possui variabilidades ligadas ao tipo de conexão e ao sistema de gerenciamento de banco de dados (SGBD) utilizado. O autor do framework já disponibiliza algumas implementações alternativas para ambos os casos (ODBC e nativo para tipo de conexão, e MySQL, SyBase e Interbase para SGBD) que podem ser escolhidas durante o reúso.

Na Figura 36 é mostrado o diagrama fornecido por Camargo representando as classes e os aspectos que compõem as características de operações persistentes e de conexão, distinguidas pelas formas geométricas com linhas tracejadas. Nessa figura, os aspectos estão sendo representados por retângulos destacados em cinza, e os relacionamentos de associação que partem dos aspectos para alguma entidade representam que o aspecto afeta a entidade

entrecortando a execução/chamada de seus métodos ou introduzindo operações por meio de declarações intertipo. Os demais relacionamentos possuem a semântica convencional da UML. Camargo utilizou um subconjunto do perfil UML-F [FON99] para evidenciar os ganchos que devem ser sobrepostos pelo reutilizador. Todos os métodos com o estereótipo *<<hook>>* são abstratos e devem ser sobrepostos pelo reutilizador em classes concretas.

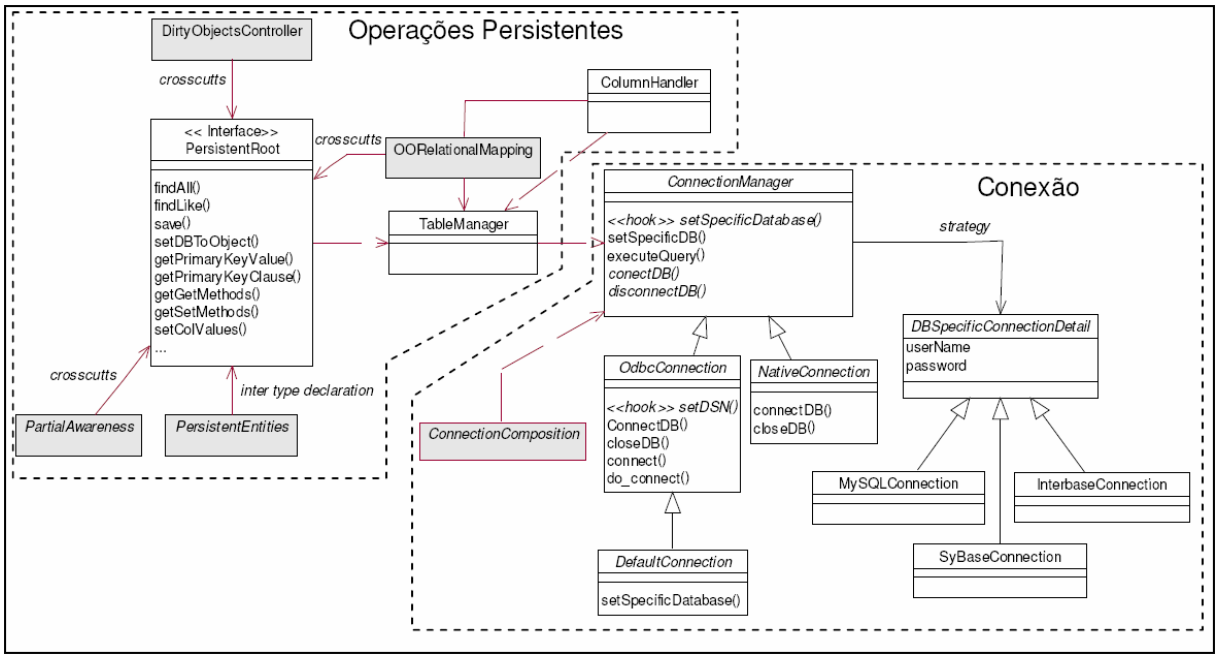


Figura 36 – Estrutura do FT de persistência.
Fonte: Camargo [CAM06].

Na parte de operações persistentes, a classe *TableManager* é responsável por montar dinamicamente as declarações SQL e executar as operações de persistência, possuindo uma dependência com o módulo de conexão. O aspecto *DirtyObjectsController* é responsável por "marcar" objetos que foram alterados em memória para que estas alterações sejam refletidas no banco em momentos apropriados. O aspecto *PersistentEntities* introduz na interface *PersistentRoot* um conjunto de atributos e métodos de persistência que será herdado pelos subtipos desta interface. Durante o reúso do FT, o reutilizador deve concretizar o aspecto abstrato *PersistentEntities*, indicando em quais classes da aplicação será introduzida uma relação de herança com *PersistentRoot*, evitando modificações invasivas nestas classes. O código de mapeamento objeto-relacional está encapsulado dentro do aspecto *OORelationalMapping*, que entrecorta os construtores dos subtipos de *PersistentRoot*, inserindo em cada objeto metadados sobre sua correspondência com o banco de dados (tabelas e colunas). Por fim, o aspecto opcional *PartialAwareness*, se presente na aplicação final, entrecorta pontos de junção adequados dos subtipos de *PersistentRoot* a fim de executar

automaticamente os métodos de inclusão e alteração, cabendo ao desenvolvedor da aplicação invocar apenas os métodos de remoção e busca. Caso não esteja presente na aplicação final, o desenvolvedor fica responsável pela invocação de todas as operações persistentes.

Quanto ao módulo de conexão, o único aspecto existente é o *ConnectionComposition*, que é responsável por entrecortar pontos de junção do código-base onde as conexões ao banco de dados serão abertas e fechadas, pelos métodos `connectDB()` e `disconnectDB()` da classe *ConnectionManager*, respectivamente. Esses pontos de junção são fornecidos pelo reutilizador pela concretização de *ConnectionComposition* e de seus conjuntos de junção abstratos *openConnection* e *closeConnection*.

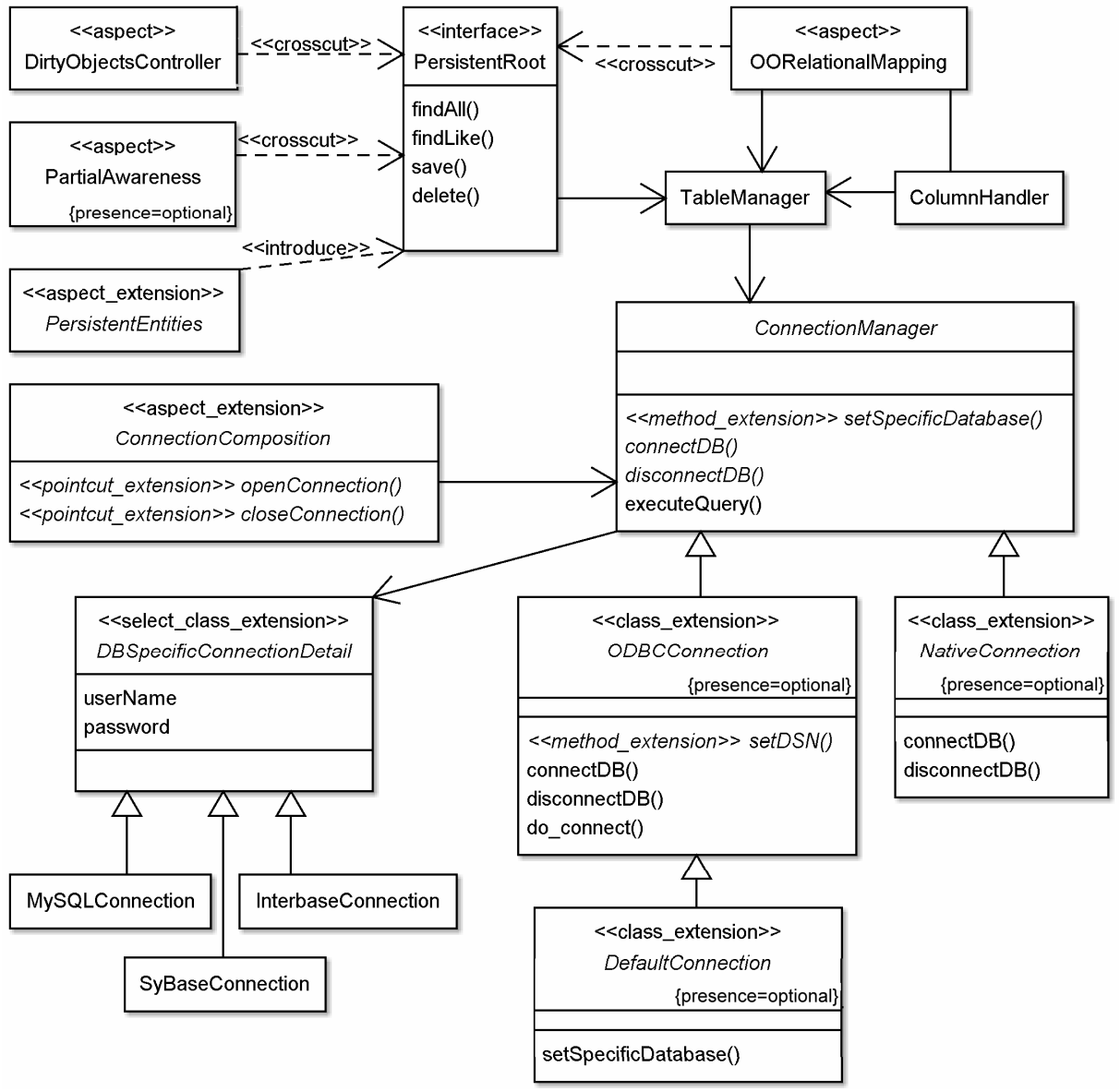


Figura 37 – FT de persistência modelado com UML-AFR e AODM.
Fonte: O autor.

Embora consiga identificar dois métodos como ponto de extensão, o modelo apresentado na Figura 36 carece de informações sobre como estes dois pontos devem ser preenchidos, além de não identificar as demais variabilidades. A Figura 37 exibe a modelagem do framework feita com UML-AFR e a versão da AODM. É possível notar que todos os pontos de extensão foram agora identificados, inclusive dando informações de como estes pontos devem ser concretizados durante o processo de reúso. Por exemplo, por causa do estereótipo *select_class_extension* na classe *DBSpecificConnectionDetail*, é possível saber que durante o processo de reúso uma de suas subclasses concretas deverá ser escolhida. Mesmo o ponto de extensão formado pelo aspecto *PersistentEntities*, que envolve declarações intertipo em sua concretização e que não são mapeadas pela UML-AFR (como já mencionado) consegue ser identificado, faltando apenas informações mais precisas de como seu reúso será realizado (que serão fornecidas pelos *cookbooks* RDL+Aspects).

O reúso desse FT possui as etapas de instanciação e de composição. Na instanciação, três variabilidades precisam ser determinadas: o tipo da consciência (total ou parcial), o tipo da conexão com o banco de dados (ODBC ou driver nativo) e o próprio banco de dados. Na etapa de composição é necessário informar quais são as classes de aplicação persistentes e em quais locais do código-base que a conexão ao banco de dados deve ser aberta e fechada.

Como já mencionado, a variabilidade de consciência é determinada pela presença ou não do aspecto *PartialAwareness*. O tipo de conexão é escolhido estendendo-se uma das classes filhas de *ConnectionManager*, informando qual SGBD será utilizado pela concretização do método *setSpecificDatabase()*, e selecionando-se a subclasse correspondente de *DBSpecificConnectionDetail*. A subclasse *DefaultConnection* define um tipo padrão de conexão, via ODBC e para o banco de dados MySQL; caso o tipo de conexão seja diferente, basta estender qualquer uma das outras subclasses. O *cookbook* RDL+Aspects que implementa essas atividades de instanciação é mostrado no Quadro 55.

A etapa de composição desse FT independe do resultado da etapa de instanciação, não havendo uma relação de ordem entre as mesmas. A composição da parte de operações persistentes consiste em estender o aspecto *PersistentEntities* e declarar quais são as classes de aplicação que devem ser persistentes, o que deve ser feito por meio de declarações intertipo da linguagem AspectJ. A composição da parte de conexão é realizada pela extensão do aspecto *ConnectionComposition* e pela concretização de seus conjuntos de junção abstratos *openConnection* e *closeConnection*, declarando em que pontos de junção a conexão ao banco de dados deve ser aberta e fechada. O Quadro 56 lista o código do *cookbook* RDL+Aspects de composição para o FT de persistência.

```
 1 instantiation cookbook PersistentCFInstantiation;
 2
 3   recipe main();
 4     // determina consciência parcial ou total
 5     define_obliviousness_level();
 6
 7     // seleciona classe específica banco de dados
 8     select_database();
 9
10     // define o tipo de conexão
11     define_connection_type();
12   end_recipe
13
14   recipe define_obliviousness_level();
15     element_choice(get_aspect("PartialAwareness"));
16   end_recipe;
17
18   recipe select_database();
19     select_class_extension(get_class("DBSpecificConnectionDetail"));
20   end_recipe;
21
22   recipe define_connection_type();
23     root : class;
24     concrete : class;
25     root := get_class("DefaultConnection");
26     specificDBMethod : method;
27     specificDBMethod := get_method(get_class("ConnectionManager"),
28                                    "getSpecificDatabase");
29     // pergunta ao reutilizador qual tipo de conexão será usado:
30     // padrão, ODBC ou nativo
31     if (element_choice(root)) then
32       // usando DefaultConnection
33       concrete := class_extension(root, ?);
34       method_extension(root, get_method(root, "setDSN"), concrete);
35     else
36       root := get_class("ODBCConnection");
37       if (element_choice(root)) then
38         // usando ODBCConnection
39         concrete := class_extension(root, ?);
40         method_extension(root, get_method(root, "setDSN"), concrete);
41         method_extension(root, specificDBMethod, concrete);
42       else
43         // usa então NativeConnection
44         root := get_class("NativeConnection");
45         concrete := class_extension(root, ?);
46         method_extension(root, specificDBMethod, concrete);
47       end_if;
48     end_if;
49   end_recipe;
50
51 end_cookbook
```

Quadro 55 – *Cookbook* de instanciação do FT de persistência.
Fonte: O autor.

```
composition cookbook PersistentCFComposition;

  recipe main();
    // compõe a parte de operações persistentes
    compose_persistent_operations()
    // compõe a parte de conexão
    compose_connection();
  end_recipe

  recipe compose_persistent_operations();
    a : aspect;
    // cria um subaspecto concreto de PersistentEntities
    a := aspect_extension(get_aspect("PersistentEntities"), ?);
    // introduz herança nas classes a serem persistidas, associando
    // as declarações intertipo ao subaspecto de PersistentEntities
    pRoot : class;
    pRoot := get_class("PersistentRoot");
    loop
      inheritance_introduction(a, pRoot, ?);
    end_loop;
  end_recipe;

  recipe compose_connection();
    a : aspect;
    pc : pointcut;
    connComp : aspect;
    connComp := get_aspect("ConnectionComposition");
    // cria subaspecto concreto de ConnectionComposition
    a := aspect_extension(connComp, ?);
    // concretiza o conjunto de junção openConnection fornecendo os
    // pontos de junção em que a conexão ao BD será aberta
    pc := pointcut_extension(connComp,
                  get_pointcut(connComp, "openConnection", a);
    loop
      add_joinpoint(pc, ?);
    end_loop;
    // concretiza o conjunto de junção closeConnection fornecendo os
    // pontos de junção em que a conexão ao BD será fechada
    pc := pointcut_extension(connComp,
                  get_pointcut(connComp, "closeConnection", a);
    loop
      add_joinpoint(pc, ?);
    end_loop;
  end_recipe;

end_cookbook
```

Quadro 56 – *Cookbook* de composição do FT de persistência.
Fonte: O autor.

O passo seguinte foi reutilizar o FT de persistência, instanciando-o e compondo-o com uma aplicação de agenda telefônica semelhante àquela utilizada no FT de segurança. Essa aplicação utiliza conexão nativa a um banco MySQL, e tem consciência total sobre as operações de persistência. Além das classes *Agenda* e *Contato*, que devem ser persistidas no banco de dados, a aplicação possui mais uma classe (*MainWindow*) contendo o método

`main()` e que fornece a interface gráfica para o usuário, contendo componentes como campos de edição e listagens. O modelo dessa simples aplicação é exibido na Figura 38.

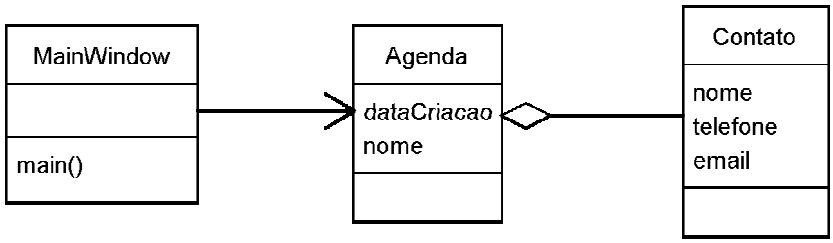


Figura 38 – Modelo da aplicação de agenda telefônica utilizado.
Fonte: O autor.

Para a execução da etapa de instanciação foram fornecidos à *Reuse Tool* o modelo do framework da Figura 37 no formato XMI e o *cookbook* de instanciação do Quadro 55. Para determinar o nível de consciência total, optou-se por não escolher o aspecto *PartialAwareness* na linha 15, fazendo com que o mesmo fosse retirado do modelo resultante. Na linha 19 foi escolhida a subclasse concreta *MySQLConnection* a fim de indicar o SGBD utilizado pela aplicação. Por fim, para definir o tipo de conexão como nativo, não foram escolhidas as classes *DefaultConnection* e *ODBCConnection*, nas linhas 31 e 37 respectivamente, fazendo com que a *Reuse Tool* executasse o bloco *else* (linhas 43-46) referentes à conexão nativa. A última interação com a ferramenta foi fornecer o nome da subclasse concreta de *NativeConnection* na linha 45 (definido como "AgendaMySQLConnection").

O *cookbook* de composição (Quadro 56), o modelo resultante da instanciação e o modelo da aplicação (Figura 38) foram fornecidos como entrada para a execução da etapa de composição. A *recipe* `compose_persistent_operations` é responsável por compor a parte de operações persistentes com a aplicação, pela extensão do aspecto *PersistentEntities* (linha 13) e das declarações intertipo de herança entre a interface *PersistentRoot* e as classes que devem ser persistentes (linhas 18-20). O nome do subaspecto foi definido como "AgendaPersistentEntities" e as classes *Agenda* e *Contato* foram passadas para o comando de introdução de herança da linha 19. Já *recipe* `compose_connection` é responsável por definir em quais pontos da aplicação a conexão ao banco de dados será aberta e fechada, pela extensão do aspecto *ConnectionComposition* (linha 29), da redefinição dos conjuntos de junção *openConnection* (linha 32) e *closeConnection* (linha 39) e da adição de pontos de junção adequados em cada um destes conjuntos de junção (linhas 34-36 e 41-43). O nome do subaspecto de *ConnectionComposition* foi definido como "AgendaConnectionComposition", e como ponto de junção, tanto para *openConnection* quanto para *closeConnection*, foi definido o método `main()` da classe *MainWindow*. Isso é possível porque o adendo associado

ao conjunto de junção *openConnection* é executado antes do ponto de junção, enquanto o adendo do conjunto de junção *closeConnection* é executado depois.

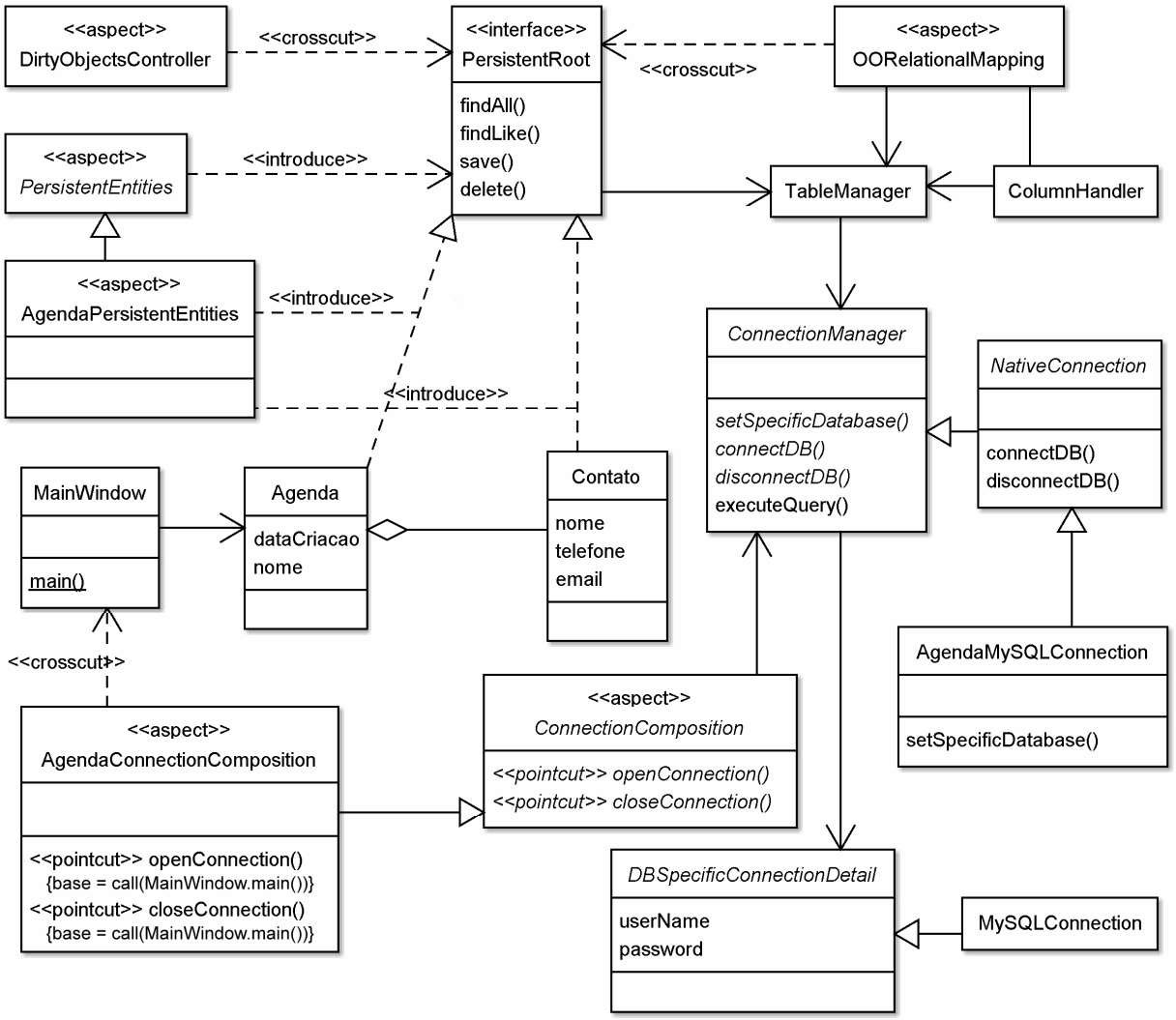


Figura 39 – Aplicação final após o reúso do FT de persistência.
Fonte: O autor.

O modelo resultante do reúso do FT de persistência, após sua instanciação e composição com a aplicação de agenda telefônica, é mostrado na Figura 39. Pode ser observado que as classes *Agenda* e *Contato* agora são persistentes, pois foi introduzida uma relação de herança entre elas e a classe *PersistentRoot* pelo aspecto concreto *AgendaPersistentEntities*. Essa declaração intertipo de herança foi configurada na ferramenta para ser representada por um relacionamento de herança simples, já que não existe uma forma adequada de representar a mesma no formato XMI. Quando o modelo foi transformado de XMI para o modelo gráfico da Figura 39 é que foi adicionada a notação não-padrão *<<introduce>>*, a fim de identificar qual aspecto realizou a introdução. Caso uma outra notação para representação da declaração intertipo seja possível no formato XMI, basta que a

ferramenta *Reuse Tool* seja configurada para interpretá-la corretamente. O entrecorte do aspecto concreto *AgendaConnectionComposition* na classe *MainWindow*, definido na etapa de composição do reúso, também pode ser visto no modelo gerado.

6.3 FRAMEWORK DE APLICAÇÃO EMBRATEC GOOD CARD

Este framework de aplicação é utilizado pela empresa Embratec Good Card [EMB06] internamente para geração de seus sistemas Java voltados para internet. Ele foi criado e desenvolvido ao longo do ano de 2006 pelo autor deste trabalho, uma vez que este é um dos arquitetos de sistemas da referida empresa. Esse FAOA utiliza tanto classes quanto aspectos para fornecer toda a infra-estrutura necessária para geração de sistemas *web*. O reúso desse framework permite o rápido desenvolvimento de sistemas de informação cadastrais, nos quais os casos de uso correspondam à manutenção de um elemento com formulários (operações de inclusão, edição, remoção e consulta).

O framework Embratec Good Card foi totalmente construído em JavaSE 5.0 e JavaEE 5.0. A arquitetura do framework é multicamada, seguindo o padrão MVC (modelo, visão e controle). Ele utiliza JSF [SUN04] para visão, o framework JBoss Seam [JBO06] para controle automático com arquivos de configuração XML, e EJBs [JCP06] (*session beans* e *entity beans*) para o modelo. O modelo, por sua vez, é dividido em três camadas:

- **Transação** – é representada por *session beans* que implementam as regras de negócio de um caso de uso, como validações de dados, restrições, cálculos, etc. Sempre que uma regra de negócio necessitar de dados do mecanismo de persistência (banco de dados, por exemplo) é feita uma requisição à camada de acesso a dados.
- **Acesso a dados** – é constituída por *session beans* que implementam as consultas e as modificações de objetos no mecanismo de persistência (banco de dados, arquivos, *web services*, memória, etc.) de modo a tornar as regras de negócio independentes deste mecanismo.
- **Entidade** – são os *entity beans* que representam objetos de negócio (dados) – como cliente, cidade, livro, etc. – e são armazenados no mecanismo de persistência escolhido para a aplicação.

Cada uma dessas camadas possui uma interface – necessária pela especificação EJB 3.0 para a definição de um EJB – e uma classe abstrata que implementa métodos padrões. No

caso das entidades, dois métodos devem ser redefinidos durante a instanciação, `getOid()` e `getVersion()`, que devem ser anotados na subclasse de forma a mapear as propriedades *oid* e *version* às colunas correspondentes da entidade. No caso dos objetos de acesso a dados (*Data Access Objects* – DAOs), as subclasses somente devem redefinir os métodos caso o comportamento destes sejam diferente do padrão implementado na classe abstrata. Por sua vez, as subclasses de transação devem possuir uma associação à classe DAO correspondente ao mesmo caso de uso, e seus métodos devem implementar as regras de negócio envolvidas.

Entrecortando a camada de transação, existem aspectos para controle de acesso (segurança) e *logging*, que são aplicados nas chamadas dos métodos de transação. Esses aspectos são implementados por interceptadores (*interceptors*) da especificação EJB 3.0 [JCP06]. Interceptadores entrecortam em tempo de execução os métodos de um *session bean*, permitindo a execução de código adicional ao entorno (antes e depois) da execução normal do método.

O aspecto de *logging* não necessita nenhuma adaptação já que simplesmente escreve informações sobre o método sendo executado na console padrão do servidor de aplicação. Já o aspecto de segurança foi implementado de forma a possibilitar que tanto a autenticação (validação de nome de usuário e senha) quanto a autorização (permissão do usuário logado para executar determinado método de negócio) possam utilizar ou um servidor LDAP ou um bando de dados para a consulta de informações. Ambos os aspectos devem ser compostos com o código do caso de uso, entrecortando todos os métodos de negócio. Como já mencionado, esse entrecorte é feito pelo uso de interceptadores, que preenchem os conjuntos de junção dinamicamente pelas anotações colocadas nos EJBs criados. Por isso, não há a necessidade de se estender o aspecto ou redefinir o conjunto de junção durante a composição, bastando adicionar os pontos de junção necessários diretamente ao conjunto de junção de cada um dos aspectos.

As informações de endereço do servidor e outras configurações para os aspectos bem como para as classes funcionais são resolvidas por uma classe específica que as lê de arquivos de configuração XML. O projeto do framework modelado com AODM e UML-AFR é exibido na Figura 40, sendo destacado quais elementos fazem parte do modelo, do aspecto de *logging* e do aspecto de segurança.

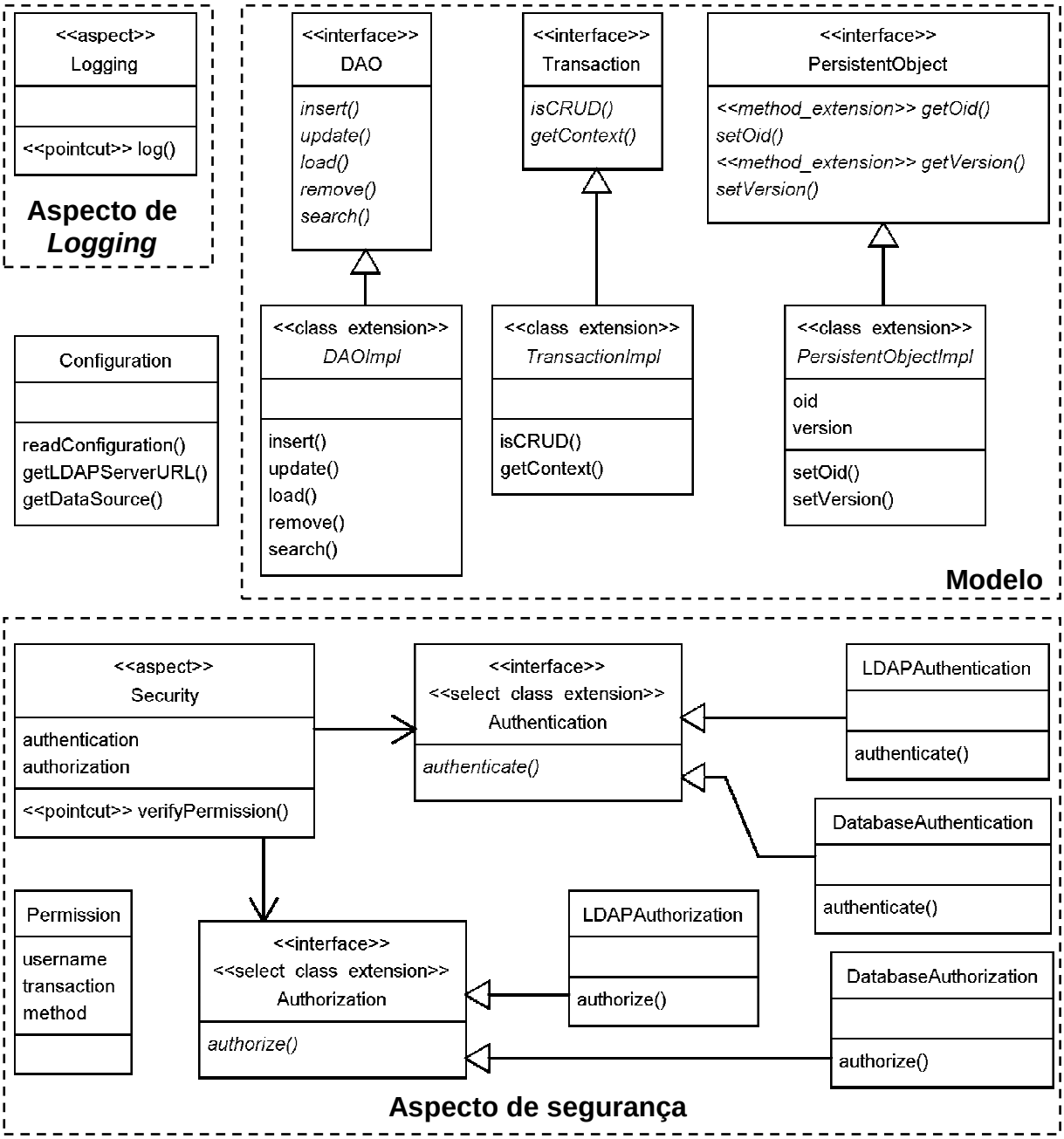


Figura 40 – FAOA Embratec Good Card modelado com UML-AFR e AODM.
Fonte: O autor.

O processo de reúso desse framework consiste nas etapas de instanciação e composição, sendo que esta última depende da primeira. Na etapa de instanciação, o desenvolvedor da aplicação deverá criar, para cada caso de uso, uma classe de acesso a dados, uma de transação e uma de entidade estendendo respectivamente as classes *DAOImpl*, *TransactionImpl* e *PersistentObjectImpl*; nessa mesma etapa ele deverá selecionar quais subclasses concretas de *Authentication* e *Authorization* são adequadas aos requisitos da aplicação (autenticação/autorização via LDAP ou banco de dados). O *cookbook* RDL+Aspects de instanciação é mostrado no Quadro 57.

```
instantiation cookbook EmbratecFRWInst;
  recipe main();
    // seleciona variabilidades do aspecto de segurança
    select_class_extension(get_class("Authentication"));
    select_class_extension(get_class("Authorization"));
    // para cada caso de uso da aplicação, cria classes necessárias
    loop
      create_use_case(?);
    end_loop;
  end_recipe;
  // cria as classes P.O, DAO e Transaction para um caso de uso
  recipe create_use_case(nome : string);
    po : class;
    dao : class;
    transaction : class;
    // busca a raiz da entidade, que é ou PersistentObjectImpl ou uma
    // subclasse de PersistentObjectImpl (no caso de generalização de
    // outra entidade
    poimpl : class;
    poimpl := get_class("PersistentObjectImpl") xo
           select_class_extension(get_class("PersistentObjectImpl"));
    // cria classe entidade (persistente)
    po := class_extension(poimpl, nome);
    method_extension(poimpl, get_method(poimpl, "getOid"), po);
    method_extension(poimpl, get_method(poimpl, "getVersion"), po);
    // adiciona demais propriedades da classe persistente
    property : string;
    loop
      property := ?;
      new_attribute(po, property);
      new_method(po, "get" + ucfirst property);
      new_method(po, "set" + ucfirst property);
    end_loop;
    // cria classe DAO
    dao := class_extension(get_class("DAOImpl"), nome + "DAO");
    // cria classe transação
    transaction := class_extension(get_class("TransactionImpl"),
                                   nome + "Transaction");
    // esse atributo referencia a dao criada, cabe ao reutilizador
    // fornecer o tipo certo
    new_attribute(transaction, ?);
    // adiciona quantos métodos de negócio forem necessários
    loop
      new_method(transaction, ?);
    end_loop;
  end_recipe;
end_cookbook
```

Quadro 57 – *Cookbook* de instanciação do FAOA Embratec Good Card.
Fonte: O autor.

Após, na etapa de composição, o desenvolvedor deverá adicionar os métodos de negócio desejados das classes de transação como pontos de junção aos conjuntos de junção *log* e *verifyPermission* dos aspectos *Logging* e *Security*, respectivamente. O *cookbook* RDL+Aspects de composição é mostrado no Quadro 58.

```
 1 composition cookbook EmbratecFRWComp requires instantiation;
 2   recipe main();
 3     // busca pointcut de log
 4     logPC : pointcut;
 5     logPC := get_pointcut(get_aspect("Logging"), "log");
 6     // busca pointcut de segurança
 7     secPC : pointcut;
 8     secPC := get_pointcut(get_aspect("Security"), "verifyPermission");
 9     // para cada caso de uso
10     transaction: class;
11     m : method;
12     loop
13       transaction := get_class(? + "Transaction");
14       // compõe aspecto de logging
15       loop
16         add_joinpoint(logPC, get_method(transaction, ?));
17       end_loop;
18       // compõe aspecto de segurança
19       loop
20         add_joinpoint(secPC, get_method(transaction, ?));
21       end_loop;
22     end_loop;
23   end_recipe
24 end_cookbook
```

Quadro 58 – *Cookbook* de composição do FAOA Embratec Good Card.
Fonte: O autor.

A fim de verificar a validade desses programas uma simples aplicação de agenda telefônica, similar às já utilizadas, foi gerada a partir do FAOA. Essa aplicação contém duas entidades – Contato e Agenda (container de contatos) – e seus respectivos casos de uso.

A primeira etapa de reúso a ser executada foi a instanciação, como definido pelo *cookbook* de composição com a construção *requires instantiation*. Nas linhas 4 e 5 do *cookbook* de instanciação foram definidos os mecanismos de autenticação e autorização, sendo escolhidas as classes *LDAPAuthentication* e *DatabaseAuthorization*, respectivamente. Após, o laço definido nas linhas 7-9 foi executado duas vezes, passando-se os valores “Contato” e “Agenda” para a *recipe* `create_use_case`. Em ambas as execuções foi selecionada como superclasse da entidade a classe *PersistentObjectImpl*, foram fornecidos os nomes das propriedades de cada entidade (*dataCriacao* e *nome* para agenda e *nome*, *telefone* e *email* para contato), e criados métodos de negócio para inclusão, alteração, exclusão e busca (*insert*, *update*, *remove*, *search* e *load*) nas classes de transação. O modelo resultante após a etapa de instanciação pode ser visto na Figura 41. Alguns atributos e métodos de classes foram omitidos para facilitar a visualização do resultado da instanciação.

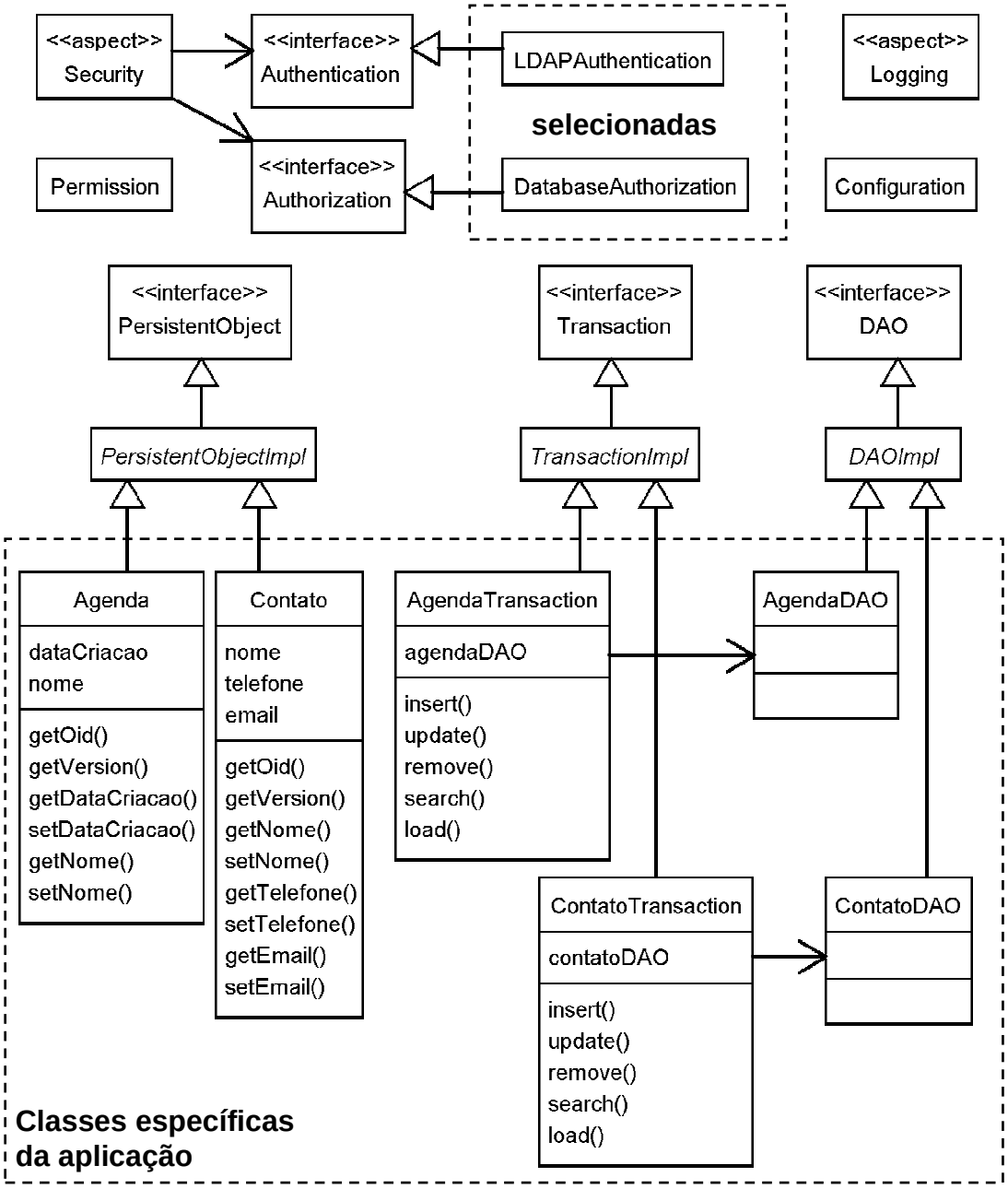


Figura 41 – FAOA Embratec Good Card instanciado (aplicação de agenda telefônica).
Fonte: O autor.

A etapa de composição foi então executada sobre o modelo instanciado do framework utilizando o *cookbook* de composição definido anteriormente. O laço definidos nas linhas 12-22 foi executado duas vezes, uma para Agenda e outra para Contato. Após informar o nome do caso de uso (entidade) na linha 13, foram fornecidos todos os métodos de negócio das classes de transação para os conjuntos de junção dos aspectos de *logging* e segurança (linhas 15-17 e 19-21, respectivamente). Somente a parte do modelo afetada pela composição é exibida na Figura 42, já que o resto permanece igual ao modelo instanciado. Os relacionamentos de entrecorte foram configurados na ferramenta *Reuse Tool* para serem

exibidos como dependências estereotipadas, sendo que os pontos entrecortados são colocados dentro de um valor identificado (*tagged value*) da dependência com o mesmo nome do conjunto de junção, conforme definido pela abordagem AODM.

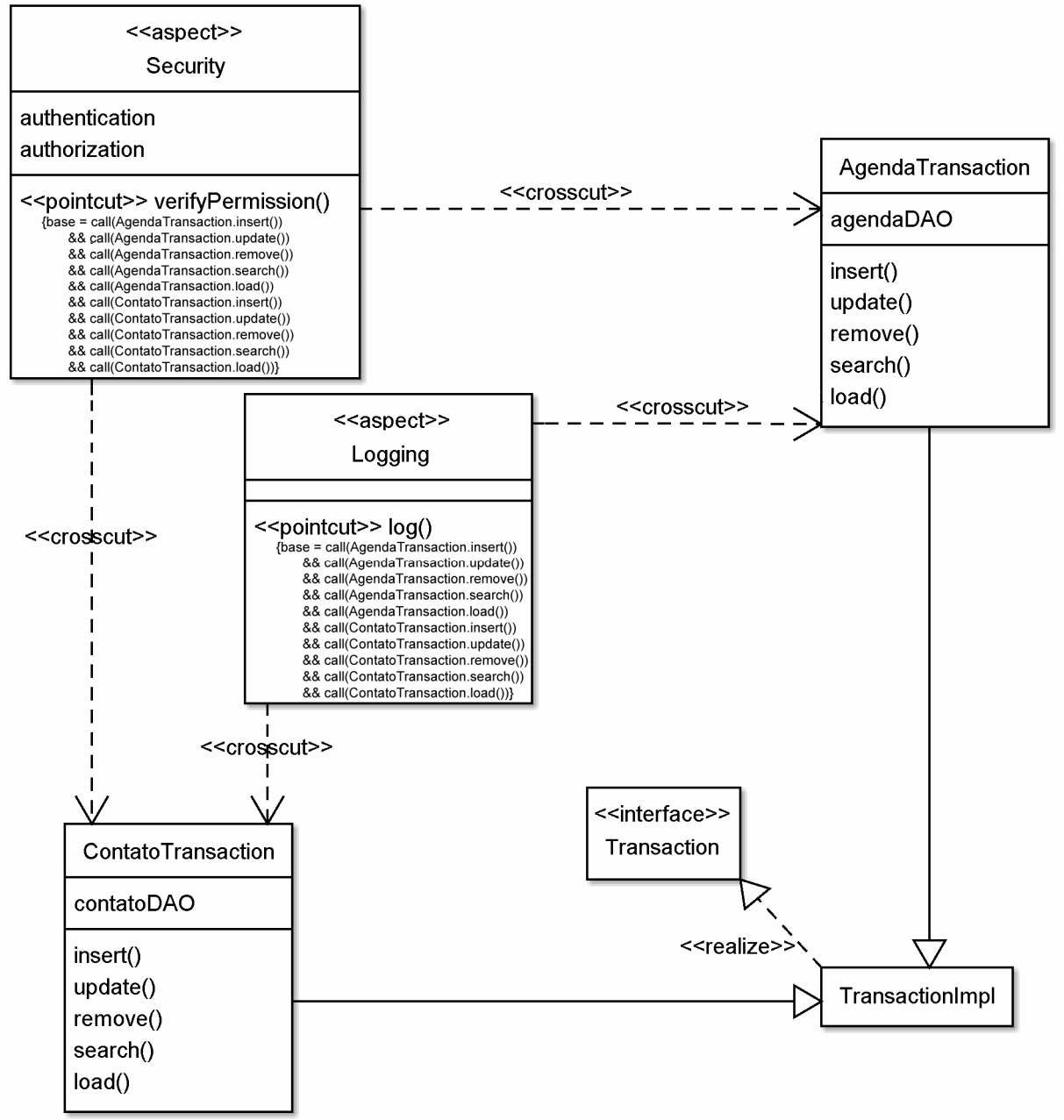


Figura 42 – Parte do modelo final da aplicação após a composição.
Fonte: O autor.

6.4 CONCLUSÕES

Nos exemplos apresentados foi possível perceber que tanto a notação UML-AFR quanto a linguagem RDL+Aspects cumpriram com êxito seus objetivos de identificar pontos de extensão e especificar as atividades de reúso dos três frameworks apresentados. Isso ocorreu tanto para frameworks transversais quanto para FAOAs, independente das etapas de reúso que o framework possua ou da dependência entre elas.

A UML-AFR conseguiu identificar os elementos de projeto que constituem pontos de extensão dos três frameworks apresentados. A UML-AFR mostrou quais classes e aspectos deveriam ser estendidos e também a forma como esta extensão deveria ser feita (criação de subelemento ou seleção). Também mostrou os conjuntos de junção e métodos que deveriam ser redefinidos a fim de fornecer informações específicas sobre a aplicação final (como quais os pontos de junção que devem ser entrecortados). A única deficiência percebida foi a falta de informações mais precisas sobre o reúso do ponto de extensão com declarações intertipo do framework de persistência, devida à falta de suporte da notação para esses casos.

A linguagem RDL+Aspects permitiu a completa especificação de todas as etapas de reúso dos frameworks, inclusive das declarações intertipo que a UML-AFR não conseguiu representar. A linguagem possibilitou a representação de atividades de instanciação e de composição, e também da dependência entre essas etapas de reúso quando houve a necessidade. A interação com o reutilizador se mostrou bastante simples, facilitando o reúso dos frameworks, principalmente com o auxílio da ferramenta *Reuse Tool*.

7 CONCLUSÕES E TRABALHOS FUTUROS

As próximas subseções exibem as conclusões alcançadas deste trabalho, sendo realçadas as principais contribuições e as vantagens e desvantagens da abordagem AFR. Também enumeram uma série de possíveis trabalhos futuros que poderão ser desenvolvidos a partir deste trabalho.

7.1 CONCLUSÕES

A utilização de frameworks orientados a objetos [PRE95][FAY99a][FAY99b] [FAY99c], ou simplesmente frameworks, encontra-se consolidada como uma das técnicas de reúso de software mais úteis e difundidas atualmente. Pelo reúso de projeto e de código, frameworks permitem a geração de sistemas inteiros de forma muito rápida e eficaz. A geração de aplicações a partir de um framework se dá por um processo de reúso, no qual seus pontos de extensão são preenchidos de acordo com os requisitos da aplicação sendo desenvolvida. No caso de FOOs, esse processo de reúso também é chamado de instanciação, pois sua execução gera uma instância (aplicação) do framework. Para que esse reúso seja feito corretamente, é necessária a completa documentação do framework, que deve responder às seguintes perguntas:

- **Para que serve?** – qual é o objetivo do framework.
- **Como ele é?** – qual é o seu projeto (estrutura, comportamento e arquitetura).
- **Quais são as possibilidades de reúso?** – identificação de seus pontos de extensão, suas características, funcionalidades e restrições.
- **Como fazer o reúso?** – especificação da seqüência de tarefas que devem ser realizadas para o correto preenchimento dos pontos de extensão.

Diversas abordagens [KRA88][LAJ94][JOH92][ORT00][CEC03][HAK01][HO 99] [FON99][OLI01][OLI04][MEN05] foram propostas com o objetivo de documentar FOOs, ou seja, de responder a essas perguntas, sendo que em sua maioria são ou documentos textuais estruturados em linguagem natural (que possui natureza intrinsecamente ambígua e interpretativa) ou são fortemente amarradas à linguagem de programação ou domínio do framework. Dessas, se destaca a abordagem RDL [OLI01][OLI04][MEN05] por oferecer um conjunto de tecnologias para: 1) a identificação de pontos de extensão (UML-FI), 2) a

especificação das atividades de reúso de maneira formal por uma linguagem de programação (RDL) e 3) a realização assistida do processo de reúso com a utilização de uma ferramenta (xFIT) que executa programas de reúso (*cookbooks*).

Com o advento da separação de interesses e da programação orientada a aspectos [KIC97a][ELR01], logo começou a ser investigado o seu uso em conjunto com frameworks, dando origem assim aos frameworks orientados a aspectos [CAM05][CAM06]. Esses frameworks possuem em sua estrutura elementos tanto da orientação a aspectos quanto da orientação a objetos, acrescentando novas formas de desenvolvimento de variabilidades. Além disso, pela própria natureza transversal dos aspectos, um framework orientado a aspectos pode possuir duas etapas distintas de reúso [CAM05]: a instanciação, assim como nos FOOs, na qual uma instância específica é gerada; e a composição, na qual o framework ou uma instância deste é composto com algum código-base previamente existente, entrecortando-o.

Graças às novas formas de construção de variabilidades e ao processo de reúso mais complexo, a necessidade de se documentar os FOAs ficou evidente. A boa documentação de um FOA facilita a sua compreensão e diminui a quantidade de erros gerada pelo processo de reúso. As abordagens desenvolvidas para FOOs conseguem documentar somente as partes orientadas a objetos dos FOAs, mas não possuem construções que permitam a correta representação das peculiaridades relacionadas à orientação a aspectos.

A notação UML-AOF [CAM04b] foi proposta com o objetivo de preencher algumas dessas lacunas deixadas pelas abordagens voltadas a FOOs. A notação UML-AOF estende a UML-F [FON99] a fim de fornecer a identificação de pontos de extensão de FOAs em seus diagramas de classes/aspectos. Ela fornece construções para diferenciar aspectos/classes da aplicação dos aspectos/classes do framework e para identificar métodos e conjuntos de junção envolvidos em um ponto de extensão. Apesar disso, a notação carece de informações mais precisas sobre o que deverá ser feito com os pontos de extensão durante o processo de reúso, e também de um mecanismo para a especificação das atividades de reúso envolvidas.

Este trabalho foi desenvolvido com o objetivo de fornecer um conjunto de tecnologias que possibilitem: 1) a identificação dos pontos de extensão de forma a já informar a maneira como o ponto de extensão será reutilizado, 2) a especificação formal das atividades de reúso envolvidas nas etapas de instanciação e composição do processo de reúso dos FOAs, e 3) uma ferramenta para possibilitar o reúso assistido por computador, sendo possível desta maneira a sistematização do processo de reúso de FOAs. Para isso, foi utilizada como base deste

trabalho a abordagem RDL [OLI01][OLI04][MEN05], que procura sistematizar o reúso de FOOs, sendo isto um requisito para a abordagem de sistematização do reúso de FOAs.

As principais contribuições da abordagem AFR apresentada neste trabalho são:

- A criação da UML-AFR e a sua incorporação ao metamodelo da UML, a partir da UML-FI, permitindo assim identificar pontos de extensão nos diagramas de projeto dos FOAs, de forma independente da linguagem de modelagem utilizada (contanto que tenha como base a UML);
- A criação da linguagem RDL+Aspects, a partir da RDL, permitindo a descrição formal da seqüência de atividades necessárias para as etapas de instanciação e composição, bem como expressar a dependência entre estas etapas;
- O desenvolvimento da *Reuse Tool*, a partir da idéia original da xFIT, possibilitando a execução assistida do processo de reúso, oferecendo desse modo maior controle ao reutilizador sobre o processo e a verificação de possíveis problemas no nível de projeto.

A notação UML-AFR permite a rápida visualização, identificação e entendimento dos pontos de extensão de um FOA em seu diagrama de classes/aspectos. Além disso, por utilizar o uso de mecanismos de extensão leve da UML, sua utilização pode ocorrer em conjunto com praticamente qualquer abordagem de modelagem orientada a aspectos baseada em UML que possua construções para aspectos e conjuntos de junção. Isso é importante pelo fato de nenhuma abordagem até hoje ter se tornado um padrão de fato, tanto na área acadêmica quanto na área corporativa.

A RDL+Aspects abstrai as atividades do processo de reúso em comandos apropriados, permitindo a criação de programas de instanciação e composição e de bibliotecas de padrões de reúso. Por manipular elementos no nível de projeto, a RDL+Aspects é independente do domínio do framework e da linguagem de programação do mesmo. Além disso, o uso da RDL+Aspects traz o processo de reúso de um FOA para a etapa de projeto de um sistema, em contraposição à realização do reúso somente na etapa de implementação, como ocorre habitualmente. Isso permite uma melhor especificação da aplicação final antes da implementação, além de aumentar a correspondência entre projeto e implementação.

A *Reuse Tool* assiste o reutilizador na execução do processo de reúso, aumentando o seu controle sobre o mesmo. Também possibilita a verificação do modelo da aplicação final, diminuindo possíveis erros que a execução manual do processo poderia causar.

Apesar das vantagens que a abordagem AFR proporciona, a mesma ainda possui algumas limitações. A notação UML-AFR não representa completamente todas as formas de

pontos de extensão orientadas a aspectos. A parte de declarações intertipo (introduções) foi omitida nesta versão da notação por não haver um mecanismo de extensão genérico bem definido, tanto nas linguagens de POA quanto nas de modelagem, para as introduções. Além disso, alguns itens foram propositadamente omitidos para não prejudicar a legibilidade do modelo, como os valores envolvidos em uma seleção ou a identificação do padrão de projeto utilizado.

Uma limitação da linguagem RDL+Aspects é a criação de novos elementos no projeto. Um elemento (classe, aspecto, método, atributo ou conjunto de junção) possui muitos outros atributos além do nome, como visibilidade (público/protegido/privado), nome do pacote, abstrato/concreto, tipo, parâmetros, etc, mas nenhum destes outros atributos consegue ser codificado dentro do *cookbook*, sendo deixado a cargo do reutilizador defini-los. Isso serve para manter a RDL+Aspects independente das linguagens de programação do framework, mas por outro lado não permite uma documentação mais precisa da forma que o elemento a ser criado deve assumir, necessitando do bom senso do reutilizador para o correto reúso do framework nessa situação.

Outra limitação da linguagem é com relação a arquivos de configuração, recurso muito utilizado por diversos frameworks existentes. Como seu foco é a manipulação de elementos no nível de projeto, esses arquivos de configuração não conseguem ser manipulados pelos comandos da RDL+Aspects, nem identificados pela UML-AFR.

O uso de aspectos também gera interferências, principalmente quando utilizados dois ou mais FOAs em conjunto. Um FOA pode ser acoplado em outro, mudando a seu comportamento de tal modo que seja necessário realizar um novo processo de reúso. As interferências também podem acontecer entre as etapas de instanciação e composição. Essas interferências podem necessitar da reexecução parcial ou total de etapas, ou de todo o processo de reúso. A reexecução parcial atualmente não é suportada pela abordagem.

7.2 TRABALHOS FUTUROS

Embora a abordagem AFR seja muito útil e consiga atingir seu principal objetivo (sistematizar o reúso de FOAs), alguns trabalhos interessantes podem ser feitos adicionalmente. A parte de declarações intertipo (introduções) poderá ser melhor explorada em futuras versões da UML-AFR, quando mecanismos bem definidos para a extensão das

introduções análogos aos conjunto de junção forem criados nas linguagens de modelagem orientada a aspectos. A própria definição de uma linguagem padrão para a modelagem de aspectos poderá tornar interessante uma maior amarração da abordagem a esta linguagem, permitindo que a expressividade da UML-AFR e da RDL+Aspects seja aumentada.

A ferramenta *Reuse Tool* pode ser alterada de modo a facilitar a distribuição de tarefas, que hoje funcionam em uma arquitetura cliente/servidor, mas sem servidor dedicado. Uma arquitetura *web* traria alguns benefícios adicionais como servidor centralizado e dedicado e acesso universal (de qualquer computador, de qualquer lugar do mundo).

Maiores investigações sobre interferências devido ao uso de aspectos também podem contribuir para o desenvolvimento da abordagem. Outra investigação que poderia ser feita é com relação às restrições de comandos disponíveis da RDL+Aspects para cada etapa de reúso. Nesta versão ambos os tipos de *cookbooks* (instanciação e composição) podem utilizar qualquer comando da linguagem. Estudos práticos em situações reais podem ser feitos para comprovar ou não a necessidade dessas restrições.

Por fim, estudos de caso podem ser feitos a fim de quantificar os benefícios trazidos pelo uso da abordagem. Para isso, é necessário o uso de alguns FOAs, documentados de diversas formas, incluindo a abordagem AFR. Após, grupos de desenvolvedores realizariam o reúso desses framework a partir das diversas documentações utilizando os mesmos requisitos para uma aplicação específica, medindo-se conceitos interessantes como tempo de desenvolvimento, custo, quantidade de erros, entre outros. A elaboração de tabelas comparativas entre essas medições conseguiria definir o quão vantajoso seria a utilização da AFR em relação a outras formas de documentação de FOAs.

REFERÊNCIAS

[ASP02] AspectWerkz. “AspectWerkz”. Capturado em: http://aspectwerkz.codehaus.org, Dezembro 2006.

[BAR04] E. Barra, G. Génova, J. Llorens. “An Approach to Aspect Modelling with UML 2.0”. In: Proc. 5th Workshop on Aspect-Oriented Modeling, Lisboa, 2004, pp. 1-7.

[BIG89] T. Biggerstaff, A. Perlis. “Software Reusability, Volume I: Concepts and Models”. Addison-Wesley, ACM Press, 1989, 425p.

[CAM04a] V. Camargo, R. Ramos, P. Masiero. “Implementação de Variabilidades em Frameworks Orientados a Aspecto desenvolvidos em AspectJ”. In: Proc. WASP’04, Brasília, 2004, pp. 1-8.

[CAM04b] V. Camargo, P. Masiero. “UML-AOF – Um Perfil UML para o Projeto de Frameworks Orientados a Aspectos”. ICMC-USP, Relatório Técnico, Abril 2004.

[CAM05] V. Camargo, P. Masiero. “Frameworks Orientados a Aspectos”. In: Proc. XIX SBES, Uberlândia, Brasil, 2005, pp. 200-215.

[CAM06] V. Camargo. “Frameworks transversais: definições, classificações, arquitetura e utilização em um processo de desenvolvimento de software”. Tese de doutorado, USP – São Carlos, 2006, 280p.

[CEC03] V. Cechticky, P. Chevalley, A. Pasetti, W. Schaufelberger. “A Generative Approach to Framework Instantiation”. Lecture Notes in Computer Science, vol. 2830, 2003, pp. 267-286.

[CHA04] C. Chavez. “A Model-Driven Approach for Aspect-Oriented Design”. Tese de doutorado, PUC-Rio, 2004, 305p.

[CLA01] S. Clarke. “Composition of Object-Oriented Software Design Models”. Tese de doutorado, Dublin City University, 2001, 285p.

[ELR01] T. Elrad, R. Filman, A. Bader. “Aspect-Oriented Programming”. Communications of the ACM, vol. 44-10, October 2001, pp. 29-32.

[EMB06] Embratec Good Card. “Embratec Good Card”. Capturado em: http://www.goodcard.com.br/, Dezembro 2006.

[FAY99a] M. Fayad, D. Schmidt, R. Johnson. “Domain-Specific Application Frameworks: Frameworks Experience by Industry”. John Wiley & Sons, 1999, 704p.

[FAY99b] M. Fayad. “Implementing Application Frameworks: Object-Oriented Frameworks at Work”. John Wiley & Sons, 1999, 729p.

[FAY99c] M. Fayad, D. Schimidt. “Building Application Frameworks: Object-Oriented Foundations of Framework Design”. John Wiley & Sons, 1999, 688p.

[FON99] M. Fontoura. “A Systematic Approach to Framework Development”. Tese de doutorado, PUC-Rio, 1999, 165p.

[FRO97] G. Froehlich, H. Hoover, L. Liu, P. Sorenson. “Hooking into Object-Oriented Application Frameworks”. In: Proc. 19th ICSE, Boston, 1997, pp. 491-501.

[GAM95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. “Design Patterns, Elements of Reusable Object-Oriented Software”. Addison-Wesley, 1995, 395p.

[HAK01] M. Hakala, J. Hautamaki, K. Koskimies, J. Paakki, A. Viljamaa. “Annotating Reusable Software Architectures with Specialization Patterns”. In: Proc. Working IEEE/IFIP Conference on Software Architecture (WICSA’01), 2001, pp. 171-180.

[HAN03] S. Hanenberg, A. Schmidmeier. “Idioms for Building Software Frameworks in AspectJ”. In: Proc. 2nd AOSD Workshop on ACP4IS, Boston, MA, March 2003, pp. 55-60.

[HAN04] Y. Han, G. Kniesel, A. Cremers. “A Meta Model and Modeling Notation for AspectJ”. In: Proc. 5th Workshop on Aspect-Oriented Modeling, Lisboa, 2004, pp. 8-15.

[HO 99] W. Ho, J. Jezequel, A. Guennec, F. Pennaneac’h. “UMLAUT: an Extendible UML Transformation Framework”. INRIA, Research Report #3775, 1999.

[JAC97] I. Jacobson, M. Griss, P. Jonsson. “Software Reuse: Architecture, Process and Organization for Business Success”. Addison-Wesley, 1997, 497p.

[JAM03] JAML. “JAML”. Capturado em: http://www.ics.uci.edu/~trungcn/jaml, Dezembro 2006.

[JBO03] JBoss. “JBoss AOP”. Capturado em: http://labs.jboss.com/portal/jbossaop, Dezembro 2006.

[JBO06] JBoss. “JBoss Seam”. Capturado em: http://labs.jboss.com/portal/jbossseam, Dezembro 2006.

[JCP06] Java Community Process. “JSR-000220 Enterprise JavaBeans 3.0”. Capturado em: http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html, Dezembro 2006.

[JOH88] R. Johnson, B. Foote. “Designing Reusable Classes”. Journal of Object Oriented Programming, vol. 1-2, June/July 1988, pp. 22-35.

[JOH92] R. Johnson. “Documenting Frameworks Using Patterns”. In: Proc. OOPSLA’92,Vancouver, Canada, 1992, pp. 63-76.

[JOH97] R. Johnson. “Components, Frameworks, Patterns”. In: Proc. Symposium on Software Reusability, USA, 1997, pp. 10-17.

[KIC96] G. Kiczales, A. Paepcke. “Open Implementations and Meta-object Protocols”. Capturado em: http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-TUT95/, Dezembro 2006.

[KIC97a] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin. “Aspect-Oriented Programming”. In: Proc. ECOOP, Finland, 1997, pp. 220-242.

[KIC97b] G. Kiczales, J. Lamping, C. Lopes, C. Maeda, A. Mendhekar, G. Murphy. “Open implementation design guidelines”. In: Proc. 19th ICSE, Boston, 1997, pp. 481-490.

[KIC01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold. “Getting started with AspectJ”. Communications of the ACM, vol. 44-10, October 2001, pp. 59-65.

[KRA88] G. Krasner, S. Pope. “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80”. Journal of Object-Oriented Programming vol. 1-3, 1988, pp. 26-49.

[LAD02] R. Laddad. “I want my AOP!, Part 1”. Capturado em: http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html, Dezembro 2006.

[LAJ94] R. Lajoie, R. Keller. “Design and reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert”. In: Proc. Colloquium on Object Orientation in Databases and Software Engineering, World Scientific, River Edge, NJ, 1994, pp. 295-312.

[LIE94] K. Lieberherr, I. Silva-Lepe, C. Xiao. “Adaptive Object-Oriented Programming Using Graph-Based Customization”. Communications of the ACM, vol. 37-5, 1994, pp. 94-101.

[MAT00] M. Mattsson. “Evolution and Composition of Object-Oriented Frameworks”. Tese de doutorado, Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby, 2000, 216p.

[MDK93] A. Mendhekar, D. Friedman. “Towards a Theory of Reflective Programming Languages”. In: Reflection’93, 1993. Capturado em: http://www2.parc.com/csl/groups/sda/publications/papers/Mendhekar-Reflect93/for-web.pdf, Dezembro 2006, 12p.

[MEN05] M. Mendonça, P. Alencar, T. Oliveira, D. Cowan. “Assisting Framework Instantiation: Enhancements to Process-Language-based Approaches”. School of Computer Science, University of Waterloo, Technical Report CS-2005-025, September 2005.

[OLI01] T. Oliveira. “Uma Abordagem Sistemática para a Instanciação de Frameworks Orientados a Objetos”. Tese de doutorado, PUC-Rio, 2001, 177p.

[OLI04] T. Oliveira, P. Alencar, I. Filho, C. Lucena, D. Cowan. “Software Process Representation and Analysis for Framework Instantiation”. IEEE Transactions on Software Engineering, vol.30-3, March 2004, pp.145-159.

[OLI07] T. Oliveira, P. Alencar, C. Lucena, D. Cowan. “RDL: A Language for Framework Instantiation Representation”. Journal os Systems and Software (JSS), 2007 (aceito).

[OMG06a] Object Management Group (OMG). “UML: Unified Modeling Language”. Capturado em: http://www.uml.org, Dezembro 2006.

[OMG06b] Object Management Group (OMG). “XMI: XML Metadata Interchange”. Capturado em: http://www.omg.org/technology/documents/formal/xmi.htm, Dezembro 2006.

[ORT00] A. Ortigosa, M. Campo, R. Salomon. “Towards Agent-Oriented Assistance for Framework Instantiation”. In: Proc. OOPSLA’00, Minneapolis, Minnesota, USA, 2000, pp. 253-263.

[PEN06a] L. Penczek, T. Oliveira. “Sistematização da instanciação de frameworks orientados a aspectos”. In: Proc. Workshop de Teses e Dissertações WTES, SBES 2006, Florianópolis, Brasil, Outubro 2006, pp. 43-48.

[PEN06b] L. Penczek, M. Mendonça, T. Oliveira. “Systemizing aspect-oriented framework reuse with AFR”. In: Proc. OOPSLA’06 Poster Session, Portland, USA, 2006, pp.665-666.

[PEN06c] L. Penczek, T. Oliveira. “AFR: an Approach to Systematize Aspect-Oriented Framework Reuse”. In: Proc. 2nd Asian Workshop on Aspect-Oriented Software Development AOAsia, ASE 2006, Tokyo, Japan, September 2006, [CDROM], 6p.

[PEN06d] L. Penczek, T. Oliveira. “RDL+Aspects: uma linguagem de processo para sistematizar o reúso de frameworks orientados a aspectos”. In: Proc. III Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos WASP’2006, Florianópolis, Brasil, Outubro 2006, pp. 129-138.

[PRE95] W. Pree. “Design Patterns for Object-Oriented Software Development”. Addison-Wesley, 1995, 268p.

[SOM04] I. Sommerville. “Software Engineering”. Addison-Wesley, 2004, 7th ed., 784p.

[SPR02] Spring Framework. “Spring AOP”. Capturado em: http://www.springframework.org, Dezembro 2006.

[STE02] D. Stein. “An Aspect-Oriented Design Model based on AspectJ and UML”. Master thesis, University of Duisburg-Essen, Germany, 2002, 203p.

[SUN95] Sun Microsystems. “Java”. Capturado em: http://java.sun.com, Dezembro 2006.

[SUN04] Sun Microsystems. “JavaEE – JavaServer Faces (JSF) Technology”. Capturado em: http://java.sun.com/javaee/javaserverfaces/, Dezembro 2006.

[SUZ99] J. Suzuki, Y. Yamamoto. “Extending UML with Aspects: Aspect Support in the Design Phase”. In: Proc. Workshop on Object-Oriented Technology, London, 1999, pp. 299-300.

[TAR01] P. Tarr, H. Ossher. “Hyper/J: multi-dimensional separation of concerns for Java”. In: Proc. 23rd ICSE, Toronto, Ontario, Canada, 2001, pp. 729-730.

[VAN01] B. Vanhaute, B. De Win, B. De Decker. “Building Frameworks in AspectJ”. In: ECOOP’2001 Workshop on Advanced Separation of Concerns, Budapest, 2001, pp. 1-6.

[XER06] Xerox Corporation. “The AspectJTM Programming Guide”. Capturado em: http://www.eclipse.org/aspectj/doc/released/progguide/, Dezembro 2006.