Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós Graduação em Ciência da Computação

Phoenix: Uma Abordagem para Reengenharia de Software Orientada a Aspectos

Vinicius Cardoso Garcia

São Carlos
Março de 2005

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós Graduação em Ciência da Computação

Phoenix: Uma Abordagem para Reengenharia de Software Orientada a Aspectos

Dissertação apresentada ao Programa de pós-graduação em Ciência da Computação do Departamento de Computação da Universidade Federal de São Carlos como parte dos requisitos para obtenção do título de Mestre na área de Engenharia de Software.

Orientador: Dr. Antonio Francisco do Prado

São Carlos
Março de 2005

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

G216pa	Garcia, Vinícius Cardoso. PHOENIX: uma abordagem para reengenharia de software orientada a aspectos / Vinícius Cardoso Garcia. -- São Carlos : UFSCar, 2005. 119 p. Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2005. 1. Engenharia de software. 2. Reengenharia orientada a objeto. 3. Sistemas de transformação de software. I. Título. CDD: 005.1 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"PHOENIX: Uma Abordagem para Reengenharia de Software Orientada a Aspectos"

VINÍCIUS CARDOSO GARCIA

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

Membros da Banca:

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

Profa. Dra. Rosana Teresinha Vaccare Braga
(ICMC/USP)

Prof. Dr. Roberto da Silva Bigonha
(UFMG)

São Carlos
Março/2005

Dedico esta dissertação a DEUS e Nossa Senhora,
a meus pais e minha irmã, cujo exemplo de honestidade e
trabalho tem sido um norteador para a minha vida,
que sempre me deram apoio nos momentos mais difíceis,
vocês me ensinaram a simplicidade de ter esperança.

Agradecimentos

Ao longo destes anos muitos foram os que contribuíram para a realização deste projeto. Gostaria de destacar aqui alguns, não necessariamente os mais importantes, mas os essenciais para a consolidação desta conquista. Inicialmente quero agradecer a Deus e a Nossa Senhora pela oportunidade e capacidade de aprender e onde eu sempre busquei forças e coragem para enfrentar os desafios e buscar meus sonhos, nunca deixando que eu me abatesse pelas quedas e tristezas durante a caminhada.

Em seguida, agradeço a todos os professores da UNIFACS (Salvador-BA) que durante a minha graduação sempre me apoiaram e me estimularam nessa caminhada, em especial Antonio Carlos Fontes Atta, Cesar Teixeira, Isaac Moreira e Vicente Manuel Moreira Júnior.

Ao professor Dr. Antonio Francisco do Prado, pela orientação e incentivo, pelos ensinamentos, amizade e principalmente pela confiança depositada em mim durante este longo período desde o projeto RHAE/CNPq até o ingresso no mestrado e pelas inúmeras discussões, nem sempre sobre os mesmos pontos de vista, mas que contribuíram significativamente para a minha formação como pesquisador e como pessoa, o meu muitíssimo obrigado.

Ao professor Dr. Roberto Bigonha e à professora Dra. Rosana Braga pelos comentários e sugestões durante a defesa da dissertação.

Ao coordenador do curso de Mestrado em Ciência da Computação, professor Dr. Luis Carlos Trevelin, pelo apoio sempre manifestado e aos docentes, Dr. Hélio Crestana Guardia, Dr. Mauro Biajiz, Dra. Rosângela Penteado e doutor Sérgio Donizetti Zorzo, pela amizade, dedicação e apoio técnico prestado neste período.

A todos os funcionários do DC que colaboraram para a realização deste trabalho, em especial Cristina Trevelin, Miriam, Dona Ofélia e Dona Vera.

A equipe do Laboratório de Engenharia de Software do Departamento de Computação da Universidade Federal de São Carlos, em especial aos colegas Adail Nogueira, Adriano Bossonaro, Ariele, Calebe Bianchini, Diogo Sobral, Edmilson Ricardo, Fabiana Assáo (Fabi), Luiza Frota e Moraes, pela ajuda em diversos momentos.

Ao amigo Daniel Lucrédio por estar sempre disposto a ajudar e discutir novas idéias e pro-

jetos, pelo companheirismo e pelos inúmeros momentos de descontração; à amiga Val Fontanette pelas inúmeras traduções e auxílio no inglês, discussões sobre reengenharia, pelo acolhimento, hospitalidade e grande ajuda durante minha chegada em São Carlos; e a João "Pelão" por partilhar diversas opiniões e ajuda em diversos momentos.

A Eduardo Kessler Piveta pelas inúmeras discussões, idéias e pela parceria em diversos trabalhos e publicações. *I shall not forget to thank Jim Coplien. He did an outstanding work, guiding me and making me understand what patterns are really about.*

Aos colegas de república pelos momentos de amizade e descontração, especialmente, Fernando Balbino, Cláudio Haruo, Alessandro Rodrigues, Cristiano *"Piafino"*, Alexandre *"Clô"*, Darley *"Biriguei"*, André, Raphael, Daniel e por último Cássio Prazeres.

Aos amigos do mestrado, em especial Dinho, Érico, Escovar, Evandro, Fabiana, Bocudo, Genta, Jessica, Jukinha, Mairum, Raquel, Taciana e outros, pelos excelentes momentos nos churrascos, festas de boas vindas e de despedidas, nas inúmeras "baladas", pelo tereré na hora do almoço e por tudo mais. Agradeço também ter participado do glorioso *SSS - Saravá Saci Soccer*, o melhor time de futsal do PPG-CC de todos os tempos, campeão do InterAnos da Computação.

Aos companheiros de todas as horas e para todas as viagens (Ouro Preto, Ibitinga, Ribeirão, Araraquara...) e por todas as outras ainda por viajar, Renato Claudino, Ricardo Argenton Ramos (RAR) e Wesley "Bel" Thereza.

Às amizades feitas em São Carlos, em especial André Rocha, Aninha e família Maricondi, Bira, Bruno (Negão), Cecília, Dani, Érika, Kamei, Roberta, Tais Calliero, Vê e Wanessa.

A Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB) e ao Departamento de Computação da UFSCar, pelo apoio financeiro para a realização deste trabalho.

Aos meus amigos-irmãos, que conviveram comigo todo este período de lutas e vitórias, sempre presentes mesmo estando a milhares de quilômetros de distância, em especial Déa, Bruninha, Duda, Fefeu, Gilberto, Luciano "Bizonho", Luciano "Padang", Mara, Márcio "Chêpa", Marcus "Manga", Paulo Marcelo, Rogério, Tiago "Xuba" e toda galera da APEBA; e à eterna galera do Villa (Fenícia e Etruska). A vocês eu tenho o seguinte a dizer: *"Às vezes a gente corre tanto pela vida e deixa passar tantas coisas bonitas... A gente corre tanto que não percebe as coisas, as flores na beirada do caminho. Às vezes a gente quer que o fruto amadureça antes do tempo. A gente tem muita pressa, mas somente o tempo sabe dar as coisas para a gente na hora certa. O tempo sabe o momento exato de dar o presente que a gente merece."*

Ao grande amigo Eduardo Almeida, companheiro desde a graduação, por participar de mais uma vitória, pelas noites em claro viajando pelo país, nos aeroportos, nas viagens sempre corridas

e cheias de entraves na hora de voltar para casa, pelas discussões sobre projetos, perspectivas e caminhos a seguir. O seu apoio foi fundamental por todo o período da graduação, mestrado e num futuro próximo, o doutorado.

A Danielle Irina Shiratori, minha namorada, pela paciência em ouvir minhas queixas e desabafos, por participar de "diálogos" sobre técnicas de reengenharia, *refactoring*, *aspect mining* e aspectos, por estar sempre presente quando eu precisei e por entender a loucura que é a vida de um estudante de mestrado, sem a sua compreensão, incentivo, carinho e apoio, a conclusão e o sucesso deste trabalho não seria possível.

Aos meus pais, a minha irmã e minha família que não pouparam esforços para me incentivar, pelo amor, carinho, e cujo suporte foi essencial para o desenvolvimento deste. É por vocês que eu procuro sempre mais. Com certeza este trabalho deve-se muito a todo esse amor. Amo todos vocês.

Finalmente, a todos aqueles que colaboraram direta ou indiretamente na realização deste.

Meus sinceros agradecimentos.

"Ao longo de minha vida, periodicamente retorno à cidade natal,
para relembrar a jornada percorrida e buscar forças naquele
chão que me viu partir."
Juscelino Kubitschek

Resumo

A constante evolução tecnológica, com as mudanças nas plataformas de hardware e software, faz com que muitos sistemas, apesar de atenderem a seus requisitos e serem completamente estáveis, tornem-se obsoletos. A reengenharia de software, integrada com técnicas modernas de engenharia reversa e avante, pode ser a solução para reconstruir esses sistemas reutilizando os conhecimentos embutidos no seu código e nas documentações disponíveis. Mesmo usando as técnicas atuais de reengenharia, alguns problemas em relação à legibilidade do código são encontrados nos sistemas reconstruídos. Até mesmo os sistemas desenvolvidos recentemente, usando técnicas modernas de análise, projeto e implementação orientada a objetos e com a utilização de padrões de software, também padecem desses problemas. Assim, motivados pelas idéias de uma melhor solução para melhorar a legibilidade do código de sistemas já construídos, propõe-se investigar uma abordagem para reconstruir tais sistemas, obtendo seu projeto e código reestruturados de forma mais legível e organizado. A Abordagem combina técnicas de engenharia reversa, modelagem, transformação de software e Programação Orientada a Aspectos para reconstruir um sistema de software a partir do seu código legado e das suas informações disponíveis. O sistema reconstruído tem sua documentação e código reestruturados segundo a Orientação a Aspectos, facilitando, assim, a sua manutenção e evolução contínua, conforme as novas tecnologias de hardware e software. Os principais mecanismos para execução da abordagem são um Sistema Transformacional e uma ferramenta CASE, que automatizam grande parte das tarefas do engenheiro de software.

Abstract

The continuous technological evolution makes many systems become obsolete due to changes in hardware and software platforms, although the fact that they attend their requirements and that they are stable. Software reengineering, integrated with modern techniques of reverse and forward engineering, can be the solution to reconstruct these systems reusing the knowledge embedded in the code and the available documentation. Even when current reengineering techniques are used, some problems regarding the legibility of the code are found in the reconstructed systems. Recently developed systems making use of modern techniques for object oriented analysis, design and implementation using software patterns also have these problems. Thus, motivated by the ideas of a better solution to improve systems codes legibility that have already been built, we intend to investigate an approach to rebuild such systems, obtaining their project and code restructured in a more readable and organized way. The approach combines reverse engineering techniques, component-based modeling, software transformation, and Aspect-Oriented Programming to reconstruct software systems from legacy code and available documentation. The code and the documentation of the reconstructed system are structured according to Aspect-Orientation, which facilitates its maintenance and continuous evolution and complies with new hardware and software technologies. The main mechanisms for the execution of the approach are a Transformational System and a CASE tool, which automates most of the software engineer tasks.

Lista de Figuras

Lista de Tabelas

Sumário

1 Introdução

1.1 Contextualização

Atualmente, existe um grande número de empresas que continuam trabalhando com sistemas implementados em linguagens de programação antigas, cuja manutenção é árdua e onerosa.

O processo de desenvolvimento e operacionalização de um produto de software passa por três fases básicas: criação, estabilização e manutenção. Na fase de criação o problema é analisado, e o sistema projetado e implementado; na fase de estabilização podem ser incluídas novas funções para atender os requisitos não funcionais do sistema e finalmente na fase de manutenção o sistema pode ser alterado para corrigir falhas, atender novos requisitos ou melhorar o seu desempenho. Nessa última fase, muitas vezes, o sistema pode ficar comprometido por manutenções estruturais não previstas no projeto original, podendo requerer sua reengenharia.

A dificuldade em atualizar esses sistemas para a utilização de novas tecnologias, como distribuição e componentização, tem motivado os pesquisadores a investigar soluções que diminuam os custos de desenvolvimento, prolonguem o tempo de vida útil do sistema, e facilitem a sua evolução e manutenção (BAXTER; PIDGEON, 1997; GAMMA et al., 1995; NEIGHBORS, 1983).

A reengenharia é uma forma de obter a reutilização de software e o entendimento do domínio da aplicação. Com ela, por meio do reuso, as informações das etapas de análise e projeto são recuperadas e organizadas de forma coerente e reutilizável.

O objetivo da reengenharia é manter o conhecimento adquirido com os sistemas legados e utilizar esses conhecimentos como base para a evolução contínua e estruturada do software. O código legado possui lógica de programação, decisões de projeto, requisitos do usuário e regras de negócio, que podem ser recuperados e reconstruídos sem perda da semântica. O software é reconstruído com inovações tecnológicas e novos requisitos podem ser adicionados para atender prazos, custos, correções de erros e melhorias de desempenho.

Os projetos recentes de reengenharia procuram integrar técnicas de engenharia reversa e avante com modernas técnicas de desenvolvimento de software, como a Orientação a Objetos (BOOCH,

1994; MEYER, 1997) em busca de um maior nível de reutilização e manutenibilidade, aumentando a produtividade do desenvolvimento e o suporte para a adição de novos requisitos.

Além dessas técnicas, vem ganhando destaque o uso de ferramentas CASE (*Computer-Aided Software Engineering*) no projeto ou reprojeto de sistemas a serem reconstruídos. Uma ferramenta CASE que possui características importantes para este projeto de pesquisa é a MVCASE. A MVCASE, objeto de outro projeto de pesquisa (ALMEIDA et al., 2002a; LUCRÉDIO, 2005), além de apoiar a especificação do sistema em linguagens de modelagem orientadas a objetos, gera código automaticamente em uma linguagem de programação orientada a objetos, a partir das especificações em alto nível, usando componentes distribuídos.

Utilizando a reengenharia na recuperação, reprojeto e reconstrução de sistemas legados, pode-se ter como resultado um sistema redocumentado e reimplementado utilizando modernas técnicas de modelagem e programação, o que possibilita maior legibilidade e melhor qualidade no código, por meio da implementação em multi-camadas, separando os códigos de interface, regras de negócio e de acesso a banco de dados.

Uma das principais técnicas de reengenharia é a da reconstrução de software usando transformações, conforme pode ser constatado em (BERGMANN; LEITE, 2002), (CASTOR et al., 2001), (LEITE; SANT'ANNA; FREITAS, 1994), (LEITE; SANT'ANNA; PRADO, 1996) e (NEIGHBORS, 1989). Dando continuidade às idéias de Reengenharia de Software desenvolveu-se, recentemente, com o apoio do RHAE/CNPq (PRADO et al., 2004), um método para auxiliar o Engenheiro de Software na reengenharia de sistemas legados usando transformações. O método de Reengenharia de Software usando Transformações (RST) (FONTANETTE et al., 2002a), (FONTANETTE et al., 2002b), (FONTANETTE et al., 2002c), (FONTANETTE et al., 2002d), (PERES et al., 2003) integra um sistema transformacional de software, uma ferramenta CASE e outras tecnologias para reconstruir sistemas legados que serão executados em uma plataforma distribuída. O autor desta dissertação inclusive, participou durante um ano como bolsista DTI deste projeto. A experiência adquirida contribuiu para o desenvolvimento do projeto de pesquisa.

O método RST cobre parte do ciclo da reengenharia de um sistema procedural, destacando-se a recuperação do projeto do código legado, permitindo o reprojeto orientado a componentes distribuídos, em uma ferramenta CASE, com reimplementação semi-automática em uma linguagem orientada a objetos, por meio da própria ferramenta CASE ou de transformações.

Entretanto, o novo código da reimplementação pode apresentar problemas, atribuídos às limitações do paradigma orientado a objetos. Técnicas como a Orientação a Objetos procuram modularizar a regras de negócio em objetos. Porém, essa abordagem de modularização é insuficiente por não proporcionar uma estrutura eficiente no desenvolvimento de sistemas mais complexos

(HARRISON; OSSHER, 1993), (KICZALES et al., 1997), (TARR et al., 1999). As limitações associadas à Orientação a Objetos, como, por exemplo, o entrelaçamento e o espalhamento de código com diferentes interesses, são apontadas atualmente como o principal problema da manutenção dos sistemas (OSSHER; TARR, 1999). Algumas dessas limitações podem ser atenuadas com a utilização dos padrões de software propostos por Gamma (1995). Entretanto, algumas extensões do paradigma orientado a objetos são propostas para solucionar estas limitações, como, por exemplo, Programação Adaptativa (LIEBERHERR; SILVA-LEPE; XAIO, 1994), Programação Orientada a Assunto (OSSHER; TARR, 1999) e Programação Orientada a Aspectos (ELRAD; FILMAN; BADER, 2001b). Essas técnicas de programação visam aumentar a modularidade, onde a Orientação a Objetos e os padrões de software não oferecem o suporte adequado.

Dessa forma, mesmo tendo-se recuperado o projeto do sistema legado em um alto nível de abstração, permitindo ao Engenheiro de Software ter uma visão legível da funcionalidade do sistema, a sua manutenção muitas vezes, ainda é uma tarefa árdua e difícil, podendo comprometer sua evolução para acompanhar as novas tecnologias de hardware e software.

Motivado por estas idéias, esta dissertação tem por objetivo pesquisar uma abordagem que ofereça recursos para contornar essas limitações, encontradas no código do sistema reimplementado de acordo com o paradigma orientado a objetos, e não contempladas atualmente nas técnicas de reengenharia de software e encontradas nos sistemas mais recentes.

O grupo de Engenharia de Software do qual o aluno e seu orientador participam desenvolveu diversas pesquisas na área de Reengenharia de Software (ABRAHÃO; PRADO, 1999), (ALVARO et al., 2003), (FUKUDA, 2000), (JESUS; FUKUDA; PRADO, 1999), (LEITE; SANT'ANNA; PRADO, 1996), (NOGUEIRA, 2002), (NOVAIS, 2002) e (PRADO, 1992). Foi destas pesquisas que se originou a idéia desta dissertação, que objetiva separar os diferentes interesses transversais, sejam eles não funcionais, como distribuição, tratamento de exceção e acesso a banco de dados, ou funcionais, como regras de negócio.

A abordagem apresentada nesta dissertação visa permitir a reconstrução de sistemas legados, por meio da sua reengenharia, fundamentada no paradigma orientado a aspectos, obtendo, assim, uma maior modularidade e um ganho considerável na manutenibilidade e na reutilização das partes do sistema reimplementado.

1.2 Organização da Dissertação

A dissertação está organizada em seis capítulos, além da seção referente às referências bibliográficas. O primeiro capítulo contém esta introdução.

No Capítulo 2 são apresentados os principais conceitos relacionados a esta dissertação. É feita uma discussão sobre as pesquisas referentes à Reengenharia de Software e a Engenharia Reversa, que é o foco desta dissertação. Um estudo sobre os principais trabalhos nestas duas áreas é apresentado, identificando os seus pontos positivos para se especificar um processo efetivo de Reengenharia de Software.

No Capítulo 3 é apresentada uma visão geral do paradigma orientado a aspectos, bem como são descritas as características fundamentais da linguagem *AspectJ* e das suas construções. São apresentadas também as direções na pesquisa de projetos de software orientados a aspectos.

No Capítulo 4 é apresentada a abordagem Phoenix para Reengenharia de Software Orientada a Aspectos, os mecanismos utilizados e uma descrição detalhada dos seus objetivos.

No Capítulo 5 é apresentada a realização de um estudo de caso e seus resultados, utilizado para validar a abordagem Phoenix.

No Capítulo 6, são apresentadas as considerações finais abordando os trabalhos correlatos, as principais contribuições e os trabalhos futuros.

E, finalmente, no Apêndice A é apresentada uma lista das publicações obtidas no período de 2003 a 2005 e o prêmio obtido pela ferramenta CASE no SBES de 2004.

2 *Reengenharia de Software*

Muitas organizações encontram-se em uma encruzilhada na sobrevivência competitiva. Encruzilhada essa criada pela revolução da informação que agora dá forma e movimenta o mundo. A informação tecnológica ganhou tanta importância que as organizações e o governo devem atualizar-se, utilizando-se das vantagens das modernas tecnologias de processamento de informação, senão ficam para trás e são ultrapassados. Infelizmente, a maioria dos sistemas de software críticos, de que as companhias e as agências de governo dependem, foram desenvolvidos há muitos anos, e a manutenção não é suficiente para manter esses sistemas atualizados.

Pior que a desatualização, Lehman e Belady provam empiricamente em (LEHMAN; BELADY, 1985) que, se nenhuma melhoria for feita, a manutenção pode degradar a qualidade do software e, conseqüentemente, sua manutenibilidade. Enquanto o número de atividades de manutenção aumenta, as deteriorações na qualidade, chamadas de *"sintomas de envelhecimento"* (*aging symptoms*) (VISAGGIO, 2001), tornam-se maiores e mais complexas para se controlar, transformando o sistema legado em uma carga que não pode ser deixada de lado, consumindo recursos vitais de seu proprietário.

Para resolver este problema, Visaggio (2001) apresenta algumas evidências experimentais de que o processo de Reengenharia pode diminuir alguns sintomas do envelhecimento. Entretanto, mesmo reconstruindo o sistema legado, não há nenhuma garantia que o novo sistema terá uma maior/melhor manutenibilidade do que antes. Por exemplo, a Orientação a Objetos, muitas vezes considerada a *"solução para a manutenção"* (BENNETT; RAJLICH, 2000), criou novos problemas para a mesma (OSSHER; TARR, 1999) e deve ser usada com muito cuidado para assegurar que a manutenção não será mais problemática do que nos sistemas legados tradicionais.

Neste contexto, este capítulo é dedicado à Reengenharia de Software, um dos tópicos essenciais para a realização da abordagem apresentada nesta dissertação. Assim, ele está organizado como se segue: a Seção 2.1 apresenta uma visão geral dos conceitos de Reengenharia de Software e engenharia reversa. Em seguida, a Seção 2.2 apresenta uma análise sobre os principais trabalhos em Reengenharia e Engenharia Reversa, tentando estabelecer um relacionamento entre elas e as novas linhas de pesquisa e, finalmente, a Seção 2.3 apresenta as considerações finais.

2.1 Reengenharia de Software e Engenharia Reversa

A Reengenharia é uma forma de obter reuso de software e entender os conceitos do domínio da aplicação sendo modificada. Seu uso facilita a compreensão de sistemas cuja documentação de análise e projeto normalmente é inexistente ou desatualizada.

De acordo com a literatura (BENNETT; RAJLICH, 2000; BIANCHI et al., 2003; OLSEM, 1998; SNEED, 1995), a Reengenharia possui quatro objetivos, a saber:

i. Aperfeiçoar a manutenibilidade: pode-se reduzir a manutenção por meio da Reengenharia de módulos menores. O problema é que não é tão simples medir o progresso nessa atividade. Na verdade, levam-se anos para obter uma clara redução em esforço desprendido em manutenção, e com isso fica difícil comprovar que este ganho foi obtido por meio da Reengenharia, mesmo porque, ele pode ter sido obtido de outras formas, como numa melhoria na equipe de manutenção ou na aplicação de métodos mais eficientes;

ii. Migração: a Reengenharia pode ser usada para migrar o software para um ambiente operacional melhor ou menos custoso. Também pode converter sistemas implementados em antigas linguagens de programação em sistemas implementados em modernas linguagens que oferecem mais recursos ou uma melhor flexibilidade;

iii. Obter maior confiabilidade: são necessários extensivos testes para demonstrar uma equivalência funcional que irá identificar antigos erros que tem afetado o software. A reestruturação revela potenciais defeitos. Esse objetivo pode ser medido prontamente pela análise de erros;

iv. Preparação para melhorias funcionais: decompor o sistema em módulos menores, aperfeiçoando a sua estrutura e isolando um módulo dos outros, tornando mais fácil realizar modificações ou adicionar novas funcionalidades sem afetar outros módulos.

2.1.1 A terminologia

A terminologia empregada na engenharia de software para referenciar as tecnologias de análise e entendimento de sistemas legados é apresentada por Chikofsky e Cross (CHIKOFSKY; CROSS, 1990), com o objetivo de racionalizar termos que já estão em uso. Os principais termos definidos e relacionados são:

- **Engenharia reversa**: é o processo de análise de um sistema para identificar os seus componentes e inter-relacionamentos, a fim de criar uma representação de outra forma ou em um nível mais alto de abstração;

- **Recuperação do projeto**: é um subconjunto da Engenharia Reversa no qual o domínio do conhecimento, informações externas e dedução são adicionadas às observações do sistema alvo para identificar as abstrações de nível mais alto em torno daquelas obtidas diretamente do exame do sistema em si. Dessa forma, tem-se um domínio mais restrito do que aquele que se tem a partir de solicitação livre de um usuário;

- **Redocumentação**: é a criação ou revisão de uma representação semanticamente equivalente a outra, com o mesmo nível de abstração que está sendo tratado. Os resultados de tal processo são usualmente considerados como visões alternativas (por exemplo, fluxo de dados, estrutura de dados e fluxo de controle) necessárias para uma exposição às pessoas;

- **Reestruturação**: é a transformação de uma representação para uma outra no mesmo nível de abstração, preservando o comportamento externo do sistema atual (funcionalidade e semântica). Reestruturação é freqüentemente uma forma aparente, tal como código alternativo, para melhorar a estrutura do sistema no sentido tradicional de projeto estruturado;

- **Engenharia avante**: é o processo de desenvolvimento de software tradicional, que inicia com abstrações de alto nível, lógicas e projetos independentes da implementação, para chegar à implementação física do sistema, ou seja, o processo de ir do projeto lógico para o projeto físico do sistema, e;

- **Reengenharia**: é a análise e a alteração de um sistema a fim de reconstituí-lo e implementá-lo em uma nova forma. Inclui algumas técnicas de Engenharia Reversa e, posterior a essa, de engenharia avante.

O relacionamento entre esses termos é mostrado na Figura 1 (CHIKOFSKY; CROSS, 1990), considerando-se que o ciclo de vida do software possui três grandes etapas: Requisitos (*Requiriments*), Projeto (*Design*) e Implementação (*Implementation*), com claras diferenças no nível de abstração. Os Requisitos tratam da especificação do problema, incluindo objetivos, restrições e regras de negócio. O Projeto trata da especificação da solução. A Implementação trata da codificação, teste e entrega do sistema em operação.

A partir dos relacionamentos da Figura 1 pode-se notar que a direção seguida pela Engenharia Avante (*Forward Engineering*), vai do nível de abstração mais alto para o mais baixo. Nota-se também que a Engenharia Reversa (*Reverse Engineering*) percorre o caminho inverso, podendo utilizar-se da Redocumentação (*Redocumentation*) para melhorar o nível de abstração. A Reengenharia (*Reengineering*) geralmente inclui uma Engenharia Reversa, seguida de alguma forma de Engenharia Avante ou Reestruturação (*Restructuring*).

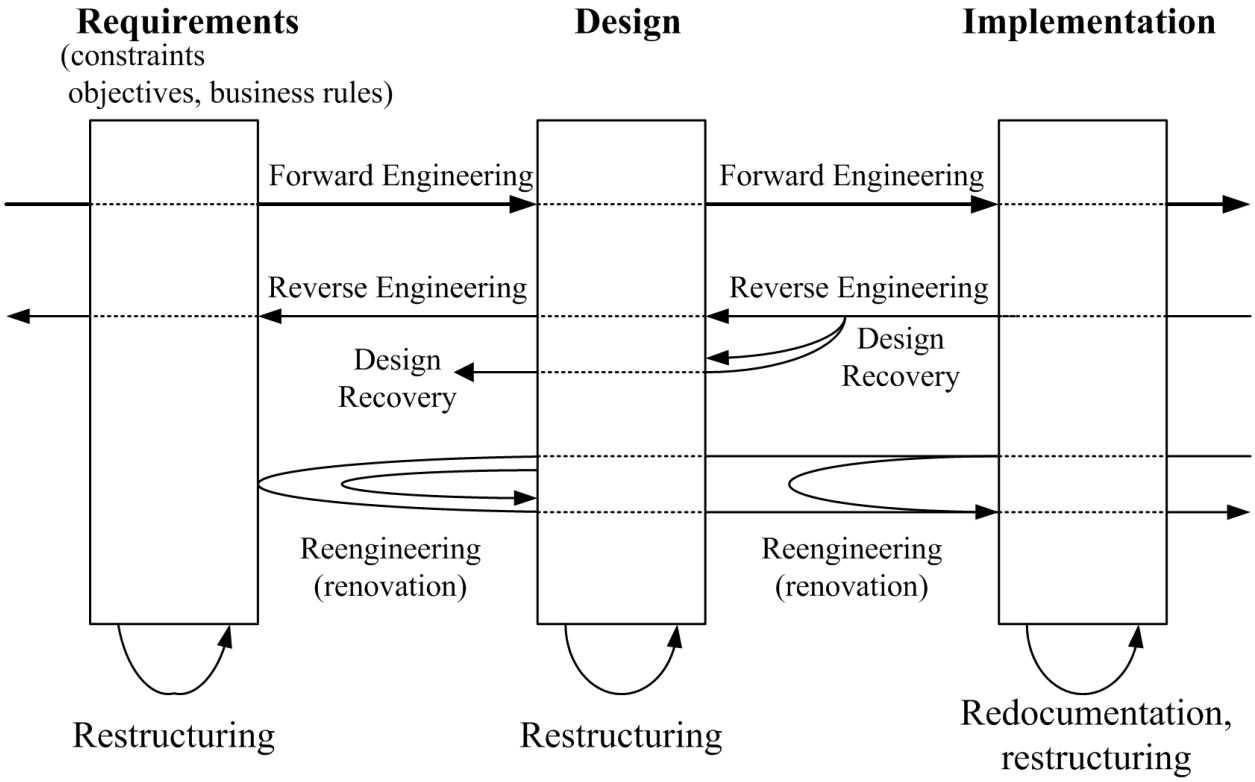


Figura 1: Ciclo de Vida do Software (CHIKOFSKY; CROSS, 1990)

Entretanto, a atividade mais importante da Reengenharia é a Engenharia Reversa, que é responsável por recuperar toda a informação que será usada no processo. Parte-se do código fonte, e obtêm-se o projeto ou a especificação do sistema do legado.

2.1.2 Engenharia Reversa

A Engenharia Reversa deve produzir, de forma mais automática possível, uma documentação que possibilite ao Engenheiro de Software um maior entendimento sobre o sistema, auxiliando assim na sua manutenção, reutilização, teste e controle de qualidade. Nos últimos dez anos, as pesquisas em Engenharia Reversa produziram um número de estudos para análise de código, incluindo a decomposição em subsistemas (UMAR, 1997), sintetização de conceitos (BIGGERSTAFF; MITBANDER; WEBSTER, 1994), projeto, programação e utilização de padrões (GAMMA et al., 1995; STEVENS; POOLEY, 1998), análise de dependências estáticas e dinâmicas (SYSTA, 1999), métricas orientadas a objetos (CHIDAMBER; KEMERER, 1994), componentização (ALVARO et al., 2003) entre outros. No geral, estas análises foram bem sucedidas ao tratar o software no nível sintático, dirigindo-se à necessidade de informações específicas e para transpor pequenas diferenças de informação.

A Figura 2 (SOMMERVILLE, 1996) mostra o processo de Engenharia Reversa.

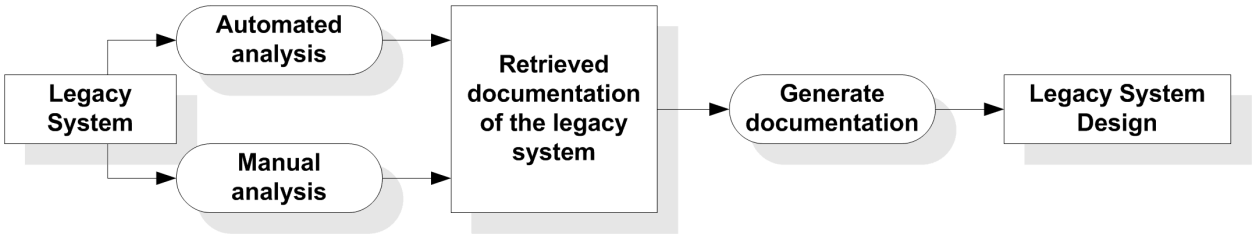


Figura 2: Processo de Engenharia Reversa (SOMMERVILLE, 1996)

O processo começa com uma fase de análise. Durante esta fase, o sistema legado pode ser analisado por meio de ferramentas para obter as informações do sistema (projeto, especificações ou diagramas arquiteturais). Os Engenheiros trabalham em nível de código para, a partir de então, recuperar o projeto do sistema. Toda informação obtida deve ser armazenada preferencialmente em diagramas que estejam em um alto nível de abstração.

Em sistemas de escala industrial, a Engenharia Reversa geralmente é aplicada por meio de abordagens semi-automatizadas usando ferramentas *CASE* (*Computer Aided Software Engineer*) para analisar a semântica do código legado. O resultado deste processo de Engenharia Reversa é importado em ferramentas de reestruturação e reconstrução e, em seguida, passa para a Engenharia Avante para finalizar o processo de Reengenharia.

2.2 Abordagens de Reengenharia de Software e Engenharia Reversa

Diversas técnicas e métodos foram propostos para enfrentar o problema da reconstrução de software. Nesta seção é apresentado um estudo sobre essas abordagens. Dentre as diversas linhas de pesquisa na engenharia reversa quatro se destacam, são elas: *Transformação Código-para-Código*, *Recuperação e Especificação de Objetos*, *Abordagens Incrementais* e *Abordagens Baseadas em Componentes*. Essas linhas de pesquisa serão apresentadas a seguir.

2.2.1 Abordagens de Transformação Código-para-Código

Essencialmente todos os tradutores de código (tradutores código-para-código e compiladores) operam por meio de transformações e refinamentos. O código fonte é transformado primeiramente na linguagem alvo em uma base de expressão-por-expressão. Vários refinamentos são aplicados a fim melhorar a qualidade do código de saída. Embora aceitável em muitas situações, esta abordagem é fundamentalmente limitada para Reengenharia devido à baixa qualidade da saída produzida. Especialmente, tende a ser insuficientemente sensível às características globais do código fonte e

demasiado sensível aos detalhes locais irrelevantes (WATERS, 1988).

O tradutor *Lisp-to-Fortran* proposto por Boyle (BOYLE; MURALIDHARAN, 1984) é baseado na abordagem transformacional. O tradutor trata um subconjunto aplicativo do *Lisp* que não inclui características difíceis de traduzir, tais como a habilidade de criar e executar novo código *Lisp*. A legibilidade não é um objetivo da tradução. Particularmente, a legibilidade da saída é abandonada em detrimento de produzir o código razoavelmente eficiente do *Fortran*. O processo de transformação é controlado pela sua divisão em um número de fases. Cada fase aplica a transformação selecionada de um pequeno conjunto. As transformações dentro de cada conjunto são escolhidas de uma maneira que não ocorram conflitos entre as transformações.

Waters (WATERS, 1988) apresenta um paradigma alternativo de tradução - abstração e reimplementação. Neste paradigma, o código fonte é primeiramente analisado a fim obter uma compreensão independente da linguagem de programação da computação executada pelo programa como um todo. Em seguida, o código é reimplementado na linguagem alvo, com base nesta compreensão. Em contraste com a abordagem da tradução padrão de transformação e refinamento, a tradução através da análise e reimplementação utiliza uma compreensão profunda, que torna possível que o tradutor crie o código alvo sem ter problemas com detalhes irrelevantes do código fonte.

A principal vantagem da tradução de código-para-código é a velocidade com que é executada, sendo mais rápida do que abordagens tradicionais, requerendo menos esforço manual. Entretanto, ainda existem algumas limitações significativas na qualidade da saída que é produzida por tradutores típicos. Isto pode claramente ser visto na tradução de código-para-código, onde a intervenção humana é requerida tipicamente a fim produzir uma saída aceitável.

2.2.2 Abordagens de Recuperação e Especificação de Objetos

A palavra-chave do final dos anos 80 e início dos anos 90 era a Orientação a Objetos. O paradigma orientado a objetos oferece algumas características desejáveis, que ajudam significativamente na melhoria do reuso de software. Esta foi a tendência predominante do software dos anos 90. De acordo com a literatura, a Orientação a Objetos deveria aumentar a manutenibilidade, reduzir a taxa de erros, aumentar a produtividade e tornar *"o mundo do processamento de dados um lugar melhor para se viver"* (MEYER, 1997).

A idéia de aplicar Engenharia Reversa orientada a objetos é que o Engenheiro de Software pode obter o projeto de um sistema existente de uma forma simplificada e com esforços limitados. Com o projeto, o Engenheiro de Software pode analisar sobre onde a modificação deve ser realizada, sua extensão, e como será mapeada no sistema existente. Além disso, o novo projeto orientado a

objetos pode servir como base para um projeto futuro de desenvolvimento. As extensões podem ser projetadas como adições fora dos sistemas existentes.

O primeiro trabalho relevante envolvendo a tecnologia orientada a objetos foi desenvolvido por Jacobson e Lindstron (JACOBSON; LINDSTROM, 1991), que apresentam uma técnica para efetuar a Reengenharia de sistemas legados, implementados em uma linguagem procedimental como *C* ou *Cobol*, obtendo sistemas orientados a objetos. Os autores mostram como realizar a Reengenharia de forma gradual, pois consideram impraticável substituir um sistema antigo por um completamente novo (o que exigiria muitos recursos). Consideram três cenários diferentes: no primeiro se faz mudança de implementação sem mudança de funcionalidade; no segundo se faz a mudança parcial da implementação sem mudança de funcionalidade; e no terceiro se faz alguma mudança na funcionalidade. A Orientação a Objetos foi usada para garantir a modularidade. Os autores utilizaram uma ferramenta *CASE* específica e defendem a idéia de que os processos de Reengenharia devem focar o uso de ferramentas para auxiliar o Engenheiro de Software. Eles também afirmam que a Reengenharia deve ser incorporada como parte do processo de desenvolvimento de software, ao invés de ser um substituto para ele.

Gall e Klösh (GALL; KLÖSCH, 1994) afirmam que a transformação de programas procedimentais em programas orientados a objetos é um processo importante para aumentar o potencial de reuso dos programas procedimentais, mas que existem problemas difíceis de se resolver no paradigma procedural, como é o caso da interconexão de módulos. Relatam que o reuso está assumindo papel relevante na produção industrial de software, pois reduz os custos de desenvolvimento de software e melhora a qualidade do produto final. Os autores propõem um processo de transformação, denominado método *COREM* (*Capsule Oriented Reverse Engineering Method*) (GALL; KLÖSCH, 1993), que usa o conhecimento do domínio da aplicação. Esse método consiste em quatro passos principais: recuperação do projeto, modelagem da aplicação, mapeamento dos objetos e transformação do sistema. Embora a aplicação de ferramentas seja útil e necessária durante o processo de transformação *COREM*, é imprescindível a aquisição de conhecimento adicional pelo Engenheiro de Software. Portanto, a automação completa desse processo de transformação não é possível, mas muitos passos do processo podem ser auxiliados por ferramentas.

Yeh, Harris e Reubenstein (YEH; HARRIS; REUBENSTEIN, 1995) propõem uma abordagem mais conservadora baseada não apenas na busca dos objetos, mas dirigida para encontrar tipos de dados abstratos (ADTs: *abstract data types*). A abordagem, chamada OBAD, é encapsulada por uma ferramenta, que usa um grafo de dependência de dados entre procedimentos e os tipos de estrutura como um ponto de início para a seleção de candidatos a ADTs. Os procedimentos e os tipos de estrutura são os nós do grafo, e as referências entre os procedimentos e os campos internos são as bordas. O conjunto de componentes conectados neste grafo formam o conjunto de candidatos a

ADTs.

Em 1995, Wilkening e outros (WILKENING et al., 1995) apresentaram um processo para efetuar a Reengenharia de sistemas legados, utilizando partes de sua implementação e projeto. Esse processo inicia-se com a reestruturação preliminar do código fonte, para introduzir algumas melhorias, como remoção de construções não-estruturadas, código "morto" e tipos implícitos. A finalidade dessa reestruturação preliminar é produzir um programa fonte mais fácil de analisar, entender e reestruturar. Em seguida, o código fonte produzido é analisado e são construídas suas representações em alto nível de abstração. Com base nessas representações, pode-se prosseguir com os passos de reestruturação, reprojeto e redocumentação, que são repetidos quantas vezes forem necessárias para se obter o sistema totalmente reestruturado. Pode-se, então, implementar o programa na linguagem destino e dar seqüência aos testes que verificarão se a funcionalidade não foi afetada.

Para transformar programas de uma estrutura procedural para orientada a objetos, pressupõem-se que os programas são estruturados e modulares, caso contrário, não podem ser transformados. Entende-se por programas estruturados, programas que não possuem nenhum *"GOTO"* como ramificações de um segmento de código a outro, por exemplo. Modulares significa que os programas estão encapsulados em uma hierarquia de segmentos de código, cada um com uma única entrada e uma única saída. Os segmentos ou procedimentos devem corresponder às operações elementares do programa. Cada operação deve ser acessível pela invocação de uma rotina de um nível mais elevado (SNEED, 1995). O outro pré-requisito é estabelecer uma árvore de chamadas de procedimentos para a hierarquia de sub-rotinas, de modo que todas as sub-rotinas sejam incluídas como parte do procedimento, com suas respectivas chamadas ou execuções.

Outro trabalho fundamental de Reengenharia orientada a objetos foi apresentado em (SNEED, 1996), em que descreve-se um processo de Reengenharia apoiado por uma ferramenta para extrair objetos a partir de programas existentes em *COBOL*. O autor ressalta a predominância da tecnologia de objetos na época, principalmente em aplicações distribuídas com interfaces gráficas, questionando a necessidade da migração de sistemas legados para essa nova tecnologia. Sneed ainda identifica obstáculos à Reengenharia orientada a objetos como, por exemplo, a identificação dos objetos, a natureza procedimental da maioria dos sistemas legados, que leva a blocos de código processando muitos objetos de dados, a existência de código redundante e a utilização arbitrária de nomes.

Assim como Wilkening e outros, Sneed assume como pré-requisitos para a Reengenharia orientada a objetos a estruturação e modularização dos programas. Entretanto, Sneed defende a existência de uma árvore de chamadas do sistema, para identificar as chamadas a procedimentos

pelo sistema. O processo de Reengenharia orientada a objetos de Sneed é composto de cinco passos: seleção de objetos, extração de operações, herança de características, eliminação de redundâncias e conversão de sintaxe. A seleção de objetos é feita com apoio da ferramenta, mas o responsável pela determinação dos objetos é o Engenheiro de Software. A extração de operações particiona o programa em sub-rotinas, removendo os segmentos referentes a um determinado objeto e substituindo-os por envio de mensagens ao objeto no qual os dados locais estão encapsulados. Foram usadas as técnicas de particionamento do programa em sub-rotinas para obtenção dos métodos e as de substituição dessas sub-rotinas por mensagens de acionamento das operações.

A transformação de programas procedimentais em programas orientados a objetos não é trivial. É um complexo conjunto de passos, um relacionamento de processo de transformação *"m:n"*, que requer a intervenção humana para definir que objetos devem ser escolhidos. Embora já existam algumas ferramentas comercialmente disponíveis para ajudar nas tarefas de Engenharia Reversa, é necessário um maior esforço para enfrentar a complexidade dos sistemas existentes, não somente por causa do seu tamanho, mas também devido à sua arquitetura interna, geralmente muito complexa.

2.2.3 Abordagens Incrementais

As abordagens apresentadas nas Seções 2.2.1 e 2.2.2 envolvem a completa reconstrução do sistema de uma vez. Por esta razão, o software deve ser "congelado" até que a execução do processo esteja terminada; ou seja, nenhuma mudança é possível durante este período. De fato, se uma mudança ou uma melhoria fossem introduzidas, o sistema legado e o candidato à atualização seriam incompatíveis, e os Engenheiros de Software teriam que recomeçar todo o processo. Esta situação causa um *loop* entre a manutenção e o processo de Reengenharia.

Para solucionar este problema, diversos autores sugerem encapsular (*wrapping*) o sistema legado, e considerá-lo como um componente caixa-preta a ser submetido à Reengenharia. Devido à natureza iterativa deste processo de Reengenharia, durante sua execução o sistema incluirá tanto os componentes da Reengenharia como os componentes legados, coexistindo e cooperando a fim de assegurar a continuidade do sistema. Finalmente, todas as atividades da manutenção, se requeridas, têm que ser realizadas tanto nos componentes da Reengenharia quanto nos legados.

O primeiro trabalho sobre processos iterativos foi proposto por Olsem (OLSEM, 1998). De acordo com o autor, sistemas legados são formados por quatro classes de componentes (Software/Aplicações, Arquivo de Dados, Plataformas e Interfaces), que não podem ser tratados da mesma forma. A Reengenharia incremental proposta pelo autor efetua o processo em componentes separados, utilizando estratégias diferentes para cada classe de componentes, diminuindo as probabili-

dades de fracasso no processo.

Outra importante contribuição do trabalho de Olsem é a proposta de duas formas de realizar a Reengenharia incremental: **com reintegração**, no qual os módulos submetidos à Reengenharia são reintegrados ao sistema, e sem **reintegração**, no qual os módulos submetidos à Reengenharia são identificados, isolados e irão se comunicar com os programas que não foram submetidos ao processo por meio de um mecanismo chamado *"Gateway"*. Com isso, o sistema resultante terá componentes que foram submetidos à Reengenharia e outros que não.

Em 2003, Bianchi e outros (BIANCHI et al., 2003) apresentaram um modelo iterativo para a Reengenharia de sistemas legados. Como inovações do processo proposto pode-se citar: a Reengenharia é gradual, isto é, é executada iterativamente em componentes diferentes (dados e funções) em fases diferentes; durante a execução do processo haverá coexistência entre os componentes legados, componentes submetidos à Reengenharia, componentes que já passaram pela Reengenharia e os novos componentes, adicionados ao sistema para satisfazer novos requisitos funcionais.

Em um outro trabalho, Zou e Kontogiannis (ZOU; KONTOGIANNIS, 2003) propõem um *framework* de transformação incremental de código, que permite que sistemas procedimentais sejam migrados para modernas plataformas orientadas a objetos. Inicialmente, o sistema é analisado gramaticalmente e um modelo em alto nível de abstração é extraído a partir do código. O *framework* introduz o conceito de um modelo unificado de domínio para uma variedade de linguagens procedimentais, tais como o *C*, *Pascal*, *COBOL* e *Fortran*. Em seguida, para manter a complexidade e o risco do processo da migração em níveis controláveis, uma técnica de *"clustering"* permite a decomposição de sistemas grandes em unidades menores e fáceis de manipular. Um conjunto de transformações de código permite a identificação de um modelo de objeto de cada unidade. Finalmente, um processo incremental consolidado permite a união dos diferentes modelos de objeto parciais em um único modelo agregado para todo o sistema.

Existem diversos benefícios associados aos processos iterativos: usando técnicas de *"divide et impera"* (*"dividir para conquistar"*), onde o problema é dividido em unidades menores, que são mais fáceis de manipular, os resultados e o retorno do investimento são imediatos e concretos; os riscos associados ao processo são reduzidos; os erros são mais fáceis de encontrar e corrigir, não põe-se todo o sistema em risco; e garante-se que o sistema continuará a funcionar mesmo durante a execução do processo, preservando a familiaridade dos mantenedores e dos usuários com o sistema (BIANCHI; CAIVANO; VISAGGIO, 2000).

2.2.4 Abordagens Baseadas em Componentes

Hoje, no topo das técnicas orientadas a objetos, está sendo estabelecida uma camada adicional no desenvolvimento de software, a baseada em componentes. Os objetivos do *"componentware"* são muito similares àqueles da Orientação a Objetos: o reuso de software deve ser facilitado e melhorado, para assim o software se tornar mais confiável e menos custoso (LEE et al., 2003).

O Desenvolvimento Baseado em Componentes não é uma idéia nova. McIlroy (MCILROY, 1969) propôs usar unidades modulares do software em 1968, e o reuso está por trás de muitos desenvolvimentos de software. A extração de componentes de software reutilizáveis do sistema é uma idéia atrativa, já que os componentes do software e seus relacionamentos incorporem uma quantidade relativamente grande de experiência do desenvolvimento passado. É necessário reutilizar essa experiência na produção de novos softwares. Essa experiência torna possível o reuso de componentes de software (CALDIERA; BASILI, 1991).

Entre os primeiros trabalhos de pesquisa nesta direção, Caldiera e Basili (CALDIERA; BASILI, 1991) exploram a extração automatizada de componentes de software reutilizáveis de sistemas existentes. Os autores propõem um processo que seja dividido em duas fases. Primeiramente, são escolhidos, no sistema existente, alguns candidatos e empacotados para um possível uso independente. Em seguida, um engenheiro com conhecimento do domínio da aplicação analisa cada componente para determinar os serviços que eles podem oferecer. A abordagem é baseada em modelos e em métricas de software.

Anos mais tarde, Neighbors (NEIGHBORS, 1996) apresentou uma pesquisa informal, realizada em um período de 12 anos, de 1980 a 1992, com entrevistas e a análise de sistemas legados, em uma tentativa de fornecer uma abordagem para a extração de componentes reutilizáveis. Embora o artigo não apresente idéias conclusivas, dá diversos indícios importantes a respeito de grandes sistemas. Conseqüentemente, as tentativas de dividir um sistema de acordo com uma destas abordagens não terão sucesso. A melhor divisão é a idéia dos subsistemas, que são encapsulados convenientemente aos programadores, aos mantenedores e aos gerentes do sistema. A etapa seguinte é extraí-los na forma de componentes reutilizáveis, que podem ser executados manualmente ou automaticamente.

Outro trabalho envolvendo componentes de software e Reengenharia pode ser visto em (ALVARO et al., 2003) no qual é apresentado um ambiente para a Reengenharia de Software Baseada em Componentes chamado *Orion-RE*. O ambiente utiliza técnicas de Reengenharia de Software e de Desenvolvimento Baseado em Componentes (DBC) para reconstruir sistemas legados, reutilizando a documentação disponível e o conhecimento embutido em seu código fonte. Um modelo de processo de software guia a utilização do ambiente por meio de uma Engenharia Reversa para re-

cuperar o projeto do sistema legado, e de uma engenharia avante no qual o sistema é reconstruído usando modernas técnicas de engenharia de software como, por exemplo, padrões, *frameworks*, técnicas de DBC e *middleware*. Como pontos positivos foi observado que a Reengenharia por meio do *Orion-RE* demanda menos tempo do que em um processo manual. Destaca-se também a utilização de um repositório distribuído para armazenamento, busca e recuperação de artefatos de software que facilita e estimula o reuso desses artefatos durante o processo, e pelo fato da abordagem ser baseada em componentes, têm-se uma melhoria na manutenibilidade.

A outra abordagem semelhante é proposta em (LEE et al., 2003), onde os autores apresentam um processo para a Reengenharia de um sistema legado orientado a objetos para um sistema baseado em componentes. Os componentes são criados com base nos relacionamentos originais das classes, que são determinados pela análise do código fonte do programa. O processo é composto de duas fases: a primeira para criar componentes básicos com o relacionamentos de composição e de herança entre as classes que constituem o sistema; e a segunda para refinar o sistema baseado em componentes intermediário usando as métricas propostas, que incluem a conectividade, a coesão e a complexidade. Finalmente, a abordagem é baseada em um modelo formal do sistema, reduzindo a possibilidade de falhas no entendimento do sistema e para permitir que operações sejam executadas corretamente.

Estas quatro abordagens são exemplos da tendência atual nas pesquisas de Engenharia Reversa, como observado por Keller e outros (KELLER et al., 1999). As abordagens Baseadas em Componentes estão sendo consideradas na Engenharia Reversa, principalmente por seus benefícios no reuso e manutenibilidade. Entretanto, ainda nota-se a falta de uma metodologia completa para a Reengenharia de sistemas legados para sistemas baseados em componentes. Mas esta falta não é restringida somente à Engenharia Reversa. Como pode ser visto em (BASS et al., 2000), os problemas encontrados quando considerando abordagens Baseadas em Componentes na Reengenharia, são somente um pequeno conjunto dos problemas relacionados à Engenharia de Software Baseada em Componentes. Enquanto esses problemas não forem solucionados, a Reengenharia terá sérias dificuldades para alcançar os benefícios associados aos componentes de software.

2.2.5 Novas Tendências de Pesquisas

A Figura 3 sumariza o estudo apresentado na seção 2.2. Analisando a linha do tempo, pode-se observar que os primeiros trabalhos estavam concentrados na transformação (tradução) de código-para-código, sem se preocupar com a legibilidade e a qualidade dos resultados. Mais tarde, com o surgimento das tecnologias orientadas a objetos, pôde-se observar um aumento no interesse com o código fonte e a qualidade da documentação, para facilitar as evoluções futuras dos sistemas.

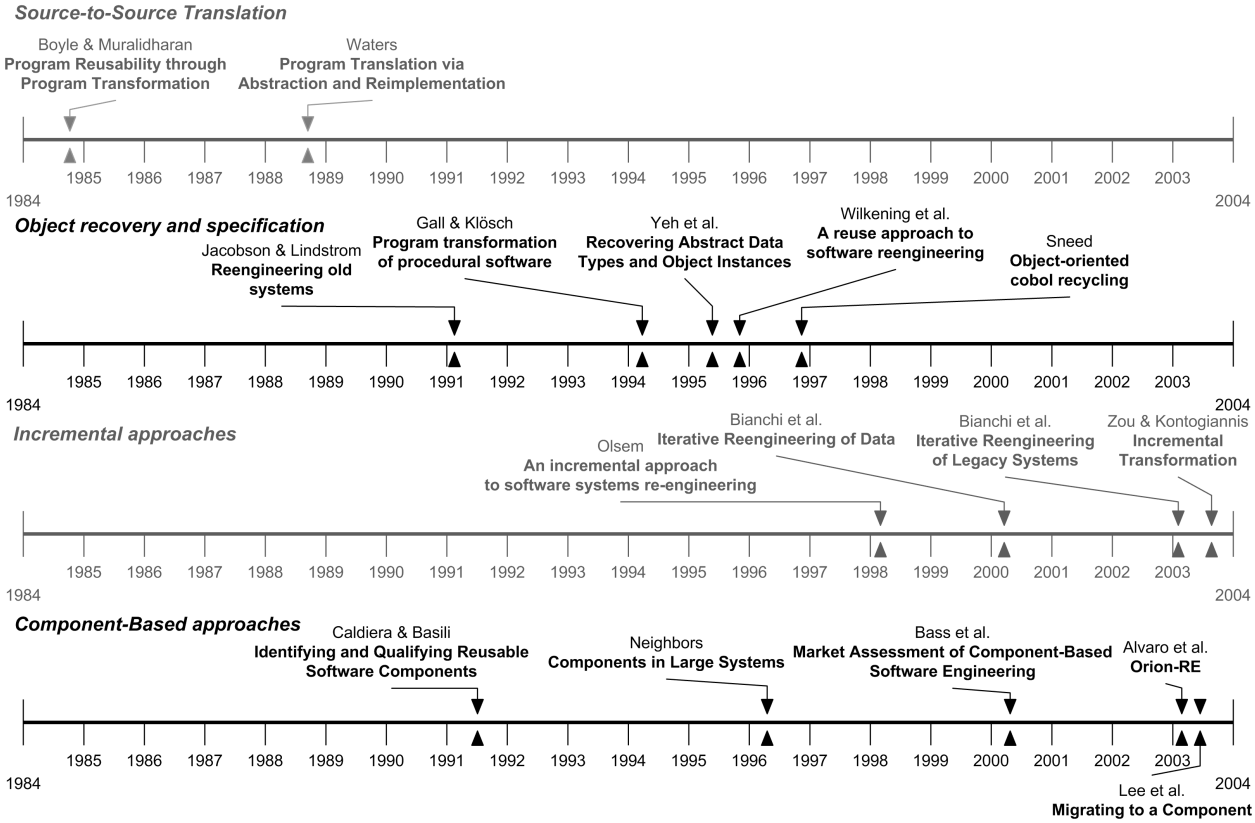


Figura 3: Linha do tempo das pesquisas sobre Reengenharia de Software e Engenharia Reversa

Entretanto, introduziu-se também os problemas relacionados à mudança de paradigma, já que a maioria dos sistemas legados eram procedimentais, sendo forçados a mudança para o paradigma orientado a objetos. Para reduzir esses problemas, as abordagens incrementais foram propostas, dando uma flexibilidade maior aos processos, permitindo a conversação entre o legado e os sistemas resultantes da Reengenharia.

As abordagens Baseadas em Componentes não acompanharam esta evolução (Código-para-Código → Orientação a Objetos → Abordagens Incrementais), sendo pesquisadas de forma esparsa pelos anos. Isto pode ser explicado pela natureza recente do Desenvolvimento Baseado em Componentes e dos problemas associados a ele, inibindo pesquisadores em seus esforços. Os esforços isolados nunca deram forma a uma tendência real na pesquisa da Reengenharia. Entretanto, este cenário está mudando, como pode ser visto na concentração dos esforços com este objetivo.

Os atuais problemas levantados e não resolvidos incluem reduzir a dispersão das funcionalidades e aumentar a modularidade, o que facilitaria a manutenção e a evolução dos sistemas resultantes da Reengenharia. Na Engenharia de Software, decompor um sistema em pequenas partes é uma forma essencial de reduzir a complexidade e garantir a sua evolução. Esta decomposição resulta na *"separação de interesses"* e facilita o trabalho paralelo, a especialização da equipe de desenvolvimento, a localização de pontos que devem sofrer modificação e conseqüentemente auxilia na

manutenção, testes sistemáticos e na garantia da qualidade (DEURSEN; MARIN; MOONEN, 2003).

Infelizmente, algumas funcionalidades existentes nos sistemas de software, como tratamento de erros ou segurança, são inerentemente difíceis de se decompor e isolar, reduzindo assim a legibilidade e a manutenibilidade destes sistemas. As técnicas da Orientação a Objetos (GAMMA et al., 1995) podem reduzir estes problemas. Contudo, a simples adoção dessas técnicas não garante a qualidade do software, já que o paradigma orientado a objetos possui algumas limitações, como o entrelaçamento e o espalhamento de código com diferentes propósitos (OSSHER; TARR, 1999). Para poucas classes e/ou poucos objetos, a utilização dessas técnicas pode funcionar adequadamente. Entretanto, a medida que o número de objetos aumenta, cresce também a quantidade de memória necessária e o número explícito de associações entre os objetos.

O Desenvolvimento de Software Orientado a Aspectos (DSOA) surgiu nos anos noventa como um paradigma direcionado a implementar interesses transversais (*crosscutting concerns*) ou mais especificamente aspectos, por meio de técnicas de geração de código para combinar (*weave*) aspectos na lógica da aplicação (KICZALES et al., 1997). De acordo com os princípios da Orientação a Aspectos, aspectos modificam componentes de software através de mecanismos estáticos e dinâmicos. Os mecanismos estáticos preocupam-se com a adição de estado e comportamento nas classes, enquanto que os dinâmicos modificam a semântica dessas classes em tempo de execução. Esses aspectos são implementados como módulos separados, de forma que fique transparente a maneira como os aspectos agem sobre os componentes funcionais, e como estes o fazem (FILMAN; FRIEDMAN, 2000).

As investigações sobre DSOA na literatura procuram determinar a extensão que pode ser usada para melhorar o desenvolvimento e a manutenção do software, ao longo das linhas discutidas por Bayer em (BAYER, 2000). O DSOA pode ser usado para reduzir a complexidade e o entrelaçamento de código; aumenta também a modularidade e o reuso, que são os principais problemas enfrentados atualmente pela Reengenharia de Software. Assim, alguns trabalhos que usam as idéias do DSOA na Reengenharia podem ser encontrados na literatura recente.

No estudo de caso realizado por Kendall (KENDALL, 2000), projetos orientados a objetos existentes são usados como ponto de início para a Reengenharia com técnicas orientadas a aspecto. Neste trabalho, o processo de Reengenharia, as técnicas, mecanismos e as etapas seguintes não são descritas em detalhes. São informados apenas os resultados comparativos entre o código orientado a objetos e o código orientado a aspectos, resultado da Reengenharia. O uso do DSOA neste estudo de caso reduziu o módulo total do sistema (classes e métodos) e o número de linhas do código. Houve uma redução de 30 métodos e 146 linhas do código no sistema orientado a aspectos [1].

[1] As regras de modelos discutidas em (KENDALL, 2000) envolvem cinco protocolos da FIPA (*Foundation for Intelligent Physical Agents*) (DICKINSON, 1997). Nos protocolos, os agentes podem pedir e receber serviços de cinco

Sant'Anna e outros (SANT'ANNA et al., 2003) apresentam algumas métricas para comparações entre os projetos e implementações orientadas a aspectos e orientadas a objetos, que podem servir para avaliar o produto do processo de Reengenharia.

Lippert e Lopes (LIPPERT; LOPES, 2000) apresentam um estudo que relata a capacidade do DSOA em facilitar a separação do código de manipulação e detecção de exceções. Como estudo de caso foi examinado um *framework*, construído em *Java*, e a parcial Reengenharia para aspectos, de parte do sistema que cuida da manipulação e detecção de exceções, usando *AspectJ*.

Atualmente como o DSOA está entrando em uma fase de inovações, novos desafios vão surgindo enquanto a tecnologia vai sendo adotada e estendida. Linguagens para apoiar o DSOA, como *AspectJ* e *AspectS*, as contribuições de diferentes grupos de pesquisa [2] e a recente integração com servidores de aplicação como *JBOSS*[3] e *BEA's WebLogic*[4] demonstram o crescimento da popularidade do DSOA.

2.3 Considerações finais

Muitas abordagens de Engenharia Reversa e Reengenharia foram propostas com o objetivo de obter representações abstratas dos sistemas legados. O objetivo é desenvolver uma imagem global do sistema, que é o primeiro passo para sua compreensão ou transformação em um sistema que melhor represente a qualidade necessária do domínio da aplicação.

Este capítulo apresentou um estudo que identificou quatro tendências da pesquisa (tradução código-para-código, recuperação e especificação de objetos, abordagens incrementais e abordagens baseadas em componentes) nas referências de Engenharia Reversa e Reengenharia, do final dos anos 80 até hoje. Os pesquisadores irão continuar a desenvolver tecnologias e ferramentas para as tarefas genéricas da Engenharia Reversa e Reengenharia, mas pesquisas futuras devem se concentrar em formas de como fazer o processo da engenharia reversa ser passível de aplicação em diversos casos, melhor definido, controlado, e otimizado (MÜLLER et al., 2000).

A tendência mais promissora nesta área são as abordagens para auxiliar na compreensão contínua do programa. A premissa de que a Engenharia Reversa necessita ser aplicada continuamente durante toda a vida do software e que é importante compreender e reconstruir o potencial do projeto e decisões arquiteturais tem maiores implicações. A integração e a adaptação da ferramenta

maneiras diferentes. Os primeiros dois protocolos envolvem pedir e receber i) serviços e ii) informações. Nos outros três protocolos, um agente negocia para serviços com os múltiplos potenciais fornecedores via iii) uma rede do contratos, iv) uma rede iterada do contrato, ou v) um leilão.

[2] `http://aosd.net`

[3] `http://www.jboss.org`

[4] `http://www.bea.org`

devem ser as preocupações centrais. As organizações devem compreender esta premissa e obter o completo suporte administrativo a fim realizar estes objetivos.

Para o futuro, segundo (MÜLLER et al., 2000), é crítico que os Engenheiros de Software respondam eficazmente a perguntas, tais como o *"Quanto de conhecimento, e em que nível de abstração, é necessário extrair de um determinado sistema, para obter as informações sobre como realizar a sua Reengenharia?"*. Assim, será necessário combinar e adaptar as tarefas de compreensão do programa aos objetivos específicos da Reengenharia.

Entretanto, as técnicas citadas não se preocupam com a separação de funcionalidades específicas, gerando assim, um código entrelaçado com interesses funcionais e não funcionais. Uma nova tendência é que a pesquisa siga em direção à área de separação de interesses, por meio do DSOA, integrado com técnicas já existentes de Engenharia Reversa e de Reengenharia de Software. Isso representará uma etapa a mais no sentido da era pós-Orientação a Objetos, para um grau de manutenibilidade e flexibilidade mais elevados. Neste contexto, o próximo capítulo aborda uma técnica que visa solucionar o problema do entrelaçamento e o espalhamento de código de diferentes interesses por todo o sistema, denominada Programação Orientada a Aspectos.

3 *Programação Orientada a Aspectos*

A Programação Orientada a Aspectos (POA) procura solucionar alguns problemas não atendidos pela Orientação a Objetos, como, por exemplo, entrelaçamento e espalhamento de código com diferentes funções. Este entrelaçamento e espalhamento tornam o desenvolvimento e a manutenção desses sistemas atividades difíceis para o Engenheiro de Software. A POA visa aumentar a modularidade, separando o código que implementa funcionalidades específicas, afetando diferentes partes do sistema, chamadas de interesses transversais (*crosscutting concerns*). Dentre as diferentes linguagens orientadas a aspectos destaca-se a linguagem *AspectJ*, que é uma extensão orientada a aspectos da linguagem Java. *AspectJ* possui construções específicas para representar os principais conceitos da POA, como aspectos, pontos de junção, conjunto de pontos de junção, adendos e declarações intertipos[1].

Neste contexto, este capítulo é dedicado à Programação Orientada a Aspectos, contribuindo assim para o embasamento teórico da abordagem apresentada nesta dissertação. Ele está organizado como se segue: na seção 3.1 são apresentados os conceitos básicos que abrangem a Programação Orientada a Aspectos. Na seção 3.2 é apresentada a especificação de uma linguagem de apoio à POA, descrevendo mais detalhadamente a linguagem *AspectJ*, alvo deste projeto de pesquisa. Na seção 3.3 são feitas considerações a respeito do projeto de software orientado a aspectos e, finalmente, na seção 3.4 têm-se as considerações finais.

3.1 Conceitos Básicos

Metodologias recentes de desenvolvimento de software (AUGUGLIARO, 2001; D'SOUZA; WILLS, 1999; RUMBAUGH; JACOBSON; BOOCH, 1999) utilizam a Orientação a Objetos (BOOCH, 1994; MEYER, 1997) em busca de um maior nível de reutilização, o que leva a um desenvolvimento de software mais rápido e a sistemas de melhor qualidade, facilitando futuras manutenções.

[1] As sugestões de tradução dos termos comumente utilizados na POA, definidas pela Comunidade Brasileira de Desenvolvimento de Software Orientado a Aspectos por meio da sua *mailing list* (*aosd-br@yahoogrupos.com.br*) e no Primeiro Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos, podem ser encontrados em: `http://twiki.im.ufba.br/bin/view/AOSDbr/TermosEmPortugues`

O software desenvolvido usando a Programação Orientada a Objetos (POO) é mais fácil de manter porque oferece mecanismos para reduzir o acoplamento e aumentar a coesão dos módulos, melhorando a manutenibilidade (PRESSMAN, 2001).

Porém, mesmo utilizando uma metodologia orientada a objetos e decompondo o sistema em um conjunto de objetos individuais, cada um representando uma funcionalidade específica, ainda existem funcionalidades que ficarão espalhadas por diferentes objetos. Isso acontece porque o paradigma orientado a objetos possui algumas limitações (OSSHER; TARR, 1999) como, por exemplo, o entrelaçamento e o espalhamento de código com diferentes propósitos. Algumas dessas limitações podem ser atenuadas com a utilização de padrões de software (GAMMA et al., 1995). Por outro lado, extensões do paradigma orientado a objetos como, por exemplo, Programação Adaptativa[2] (LIEBERHERR; SILVA-LEPE; XAIO, 1994), Programação Orientada a Assunto[3] (OSSHER; TARR, 1999) e Programação Orientada a Aspectos[4] (ELRAD; FILMAN; BADER, 2001b), procuram solucionar as limitações do paradigma orientado a objetos, aumentando a modularidade, na qual a Orientação a Objetos e os padrões de software não oferecem o suporte adequado.

A POA baseia-se no princípio de que sistemas computacionais podem ser mais bem desenvolvidos pela separação de funções específicas, afetando diferentes partes do sistema, denominadas interesses transversais (*crosscuting concerns*) (ELRAD; FILMAN; BADER, 2001b) e (KICZALES et al., 1997).

Gregor Kiczales e outros (1997) foi quem primeiramente classificou esses interesses transversais como *"aspectos"*, e, conseqüentemente, quem criou a POA. Estes interesses podem ser tanto funcionais, relacionados a regras de negócios, quanto não-funcionais, como gerenciamento de transação, distribuição e persistência em banco de dados. Com essa separação e o aumento da modularidade proposta pela POA, o sistema implementado fica mais legível, o que contribui para o seu projeto e a sua manutenção.

Em (CZARNECKI; EISENECKER; STEYAERT, 1997), os autores afirmam que a resolução de um problema é o processo de capturar os domínios apropriados, dividí-los em problemas menores e recompor as soluções de maneira que satisfaçam às especificações originais. A partir dessa idéia, os autores apresentam uma outra definição de aspecto, diferente da apresentada inicialmente por Kiczales. Segundo eles, um aspecto é uma representação parcial de alguns conceitos relacionados a uma questão. Um conceito é algo que não pode ser capturado dentro de uma unidade modular. No caso da Orientação a Objetos são as classes, enquanto que na programação procedural, as unidades são os procedimentos e as funções (HIGHLEY; LACK; MYERS, 1999).

[2]Do inglês, *Adaptive Programming*.
[3]Do inglês, *Subject-Oriented Programming*.
[4]Do inglês, *Aspect-Oriented Programming*.

Elrad, Filman e Bader (2001b) ilustram, por meio do diagrama de classes apresentado na Figura 4, como um determinado interesse não-funcional pode se apresentar entrelaçado entre as classes de um sistema, sendo assim um **interesse transversal** às demais funcionalidades da classe. Esse diagrama de classes, usando a notação UML (*Unified Modeling Language*) (RUMBAUGH; JACOBSON; BOOCH, 1999), mostra o projeto de um editor de figuras que possui duas classes concretas: *"Point"* e *"Line"*, que estão com suas funcionalidades bem encapsuladas, segundo os princípios da POO. Porém, existe a necessidade de se adicionar uma nova funcionalidade que irá atualizar o *display* toda vez que um dos elementos do tipo *"Figure"* for movido, ou seja, sempre que os valores da posição do elemento em um plano cartesiano (eixos X e Y) forem alterados.

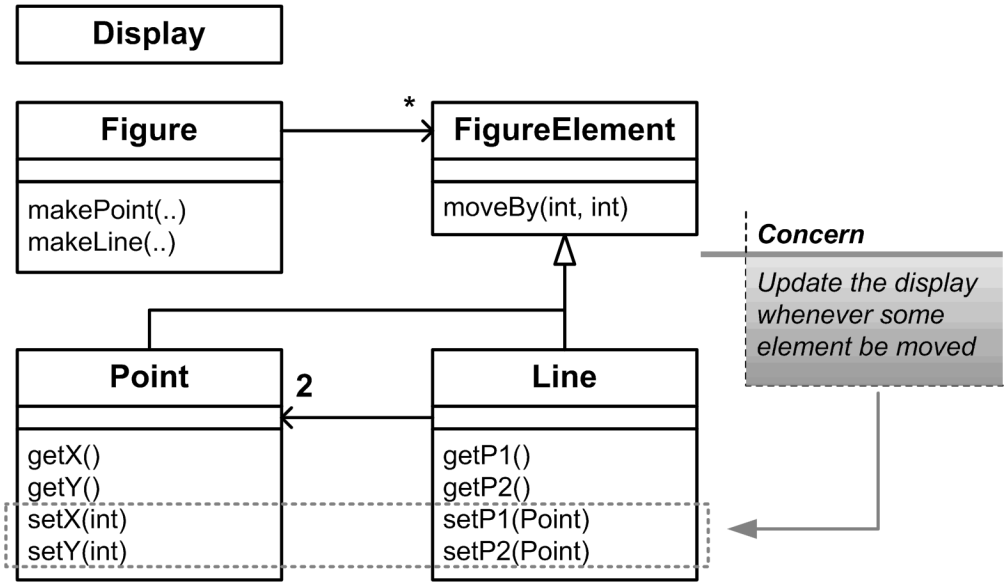


Figura 4: Editor de Figuras (ELRAD; FILMAN; BADER, 2001b)

Uma solução seria criar um método *"displayUpdating()"* responsável pela atualização do *display*, e fazer a sua invocação após o código dos métodos referentes à manipulação da posição do elemento. A implementação do mecanismo de atualização afeta métodos de diferentes classes. Em destaque, pela linha tracejada, estão os métodos que serão afetados por essa modificação, tornando o código funcional da classe entrelaçado com o código desse novo interesse recém adicionado e, este, por sua vez, espalhado por alguns métodos funcionais das classes (os métodos ‘‘`setX(int)`’’, ‘‘`setY(int)`’’, ‘‘`setP1(Point)`’’ e ‘‘`setP2(Point)`’’).

Por meio do uso da POA pode-se encapsular esse novo interesse em uma unidade modular denominada aspecto, de forma que o código que implementará essa atualização do *display* não afetará o código que trata da movimentação dos elementos.

A separação dos diferentes interesses que estão espalhados pelo sistema, nas suas unidades modulares, facilita a manutenibilidade garantindo também a evolução, compreensão e o reuso de artefatos do sistema (ELRAD; FILMAN; BADER, 2001b).

Embora possa parecer que a POA é uma evolução da POO, na verdade, ela é um paradigma para ser utilizado em conjunto com a POO, assim como também pode ser utilizado em conjunto com o paradigma procedural, já que a programação procedural também pode apresentar interesses transversais (HIGHLEY; LACK; MYERS, 1999). Além disso, a POA, atuando em conjunto com a POO, aumenta o grau de reuso das classes à medida que busca solucionar o problema dos interesses transversais.

A POA é caracterizada pela implementação de uma série de interesses fundamentais, em uma dada linguagem. Estes interesses transversais são introduzidos no sistema por meio de uma linguagem orientada a aspectos. A Figura 5 mostra o processo geral da POA.

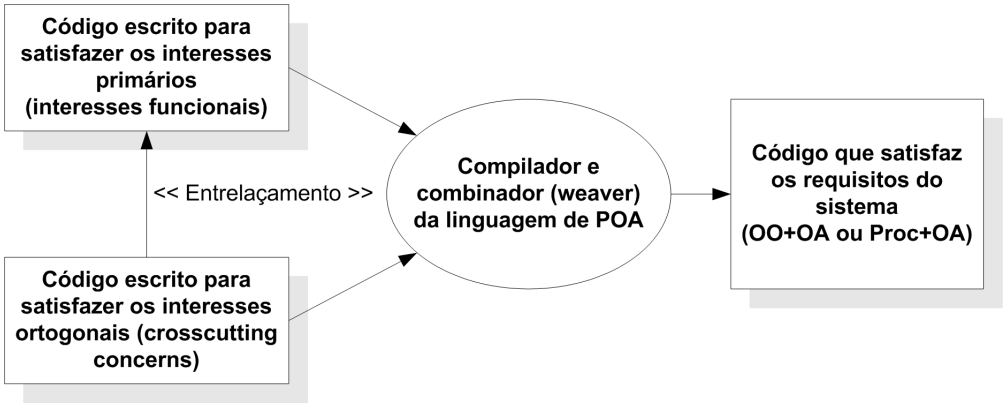


Figura 5: Processo da Programação Orientada a Aspectos (GRADECKI; LESIECKI, 2003)

O código de apoio, desenvolvido usando uma linguagem orientada a aspectos é utilizado na implementação dos interesses transversais, para então serem combinados (*weaved*) com o código denominado primário, que pode ser procedural (*Proc*) ou orientado a objetos (*OO*), para então gerar o código executável que satisfaça os requisitos do sistema.

O contraste entre o processo tradicional e o processo de combinação aspectual (*aspect weaving*) é apresentado na Figura 6.

O processo tradicional é descrito na Figura 6(A), no qual o código é submetido a um compilador, e este gera o código executável. Na POA, conforme a Figura 6(B), os aspectos são especificados separadamente, e, em seguida, o código dos aspectos é combinado com o código tradicional, por meio de um combinador de aspectos (*aspect weaver*), para, finalmente, ser submetido ao compilador que vai gerar o código executável.

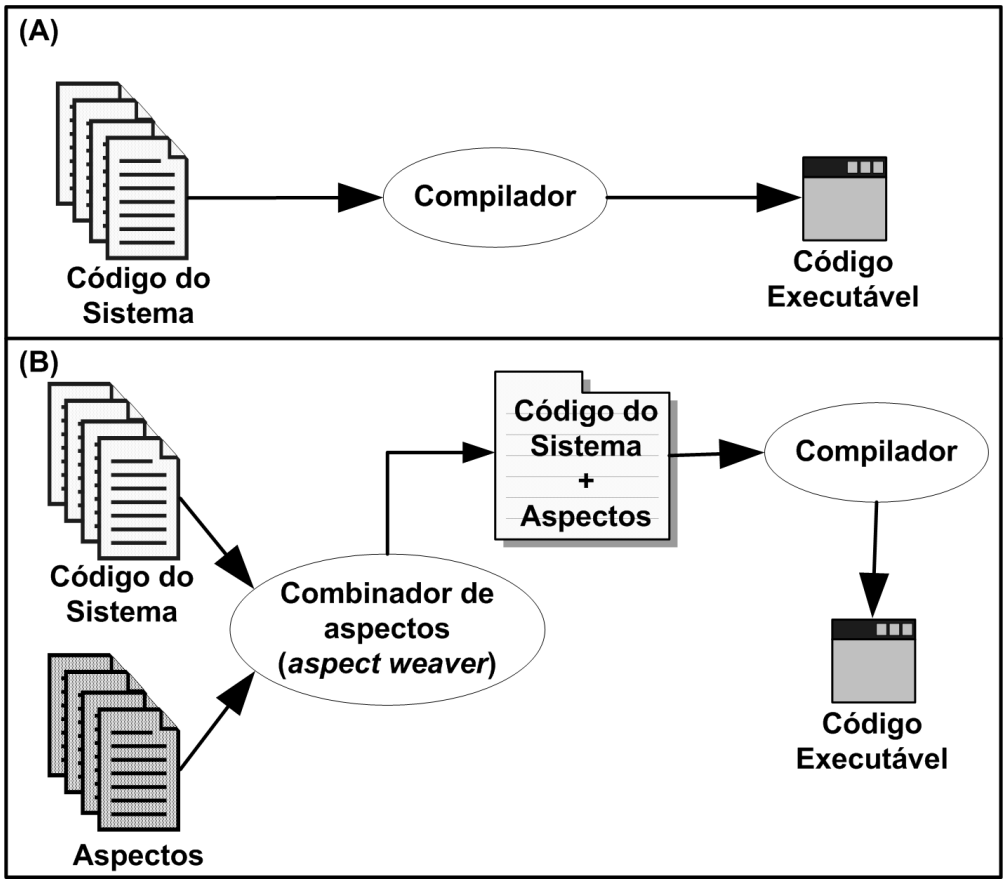


Figura 6: Compilação x Combinação (weaving)

3.2 Linguagem Orientada a Aspectos

Para implementar a separação dos interesses transversais, as linguagens orientadas a aspectos possuem um conjunto de elementos básicos na sua especificação, a saber (ELRAD et al., 2001a), (GRADECKI; LESIECKI, 2003):

- **pontos de junção (*join points*):** Um ponto de junção é um local bem definido no código no qual um interesse vai interceptar o fluxo de execução do sistema. Um ponto de junção pode ser definido por uma invocação de métodos, invocação de construtores, execução de manipuladores de exceção ou outros pontos na execução do sistema;

- **uma dada linguagem para identificar os pontos de junção:** Visto que um ponto de junção é um local bem definido no código do sistema, é necessário ter um construtor na linguagem orientada a aspectos que possa identificar esse ponto de junção. Esta construção é denominada conjunto de pontos de junção (*pointcut*);

- **uma forma de especificar o comportamento de um conjunto de pontos de junção:** A partir do momento em que um conjunto de pontos de junção é uma construção da linguagem

orientada a aspectos para identificar o ponto de junção no código do sistema, é necessário especificar a forma como os aspectos irão afetar o comportamento do sistema original. Existem três formas de afetar o comportamento: antes da execução, após a execução ou no lugar da execução do ponto de junção;

- **uma maneira de encapsular os componentes:** Após definir os construtores básicos de uma linguagem orientada a aspectos, é necessário que exista uma forma de encapsular esses elementos para depois introduzí-los no sistema. Para isso, foi especificada uma unidade modular da POA, chamada aspecto, que objetiva manipular e encapsular os pontos de junção, conjunto de pontos de junção e a forma como o comportamento do sistema será afetado.

Existem diferentes linguagens[5] para a implementação de sistemas orientados a aspectos, tais como: *AspectJ*, *Hyper/J*, *AspectR*, *AspectS*, *Apostle*, *AspectC* e *AspectC++*, porém, apenas *AspectJ* será comentada nesta dissertação por ser a que se apresenta mais madura e a que está em foco em eventos e publicações internacionais como *ECOOP* [6], *OOPSLA* [7], *Communications of ACM*, *IEEE Transactions on Software Engineering*, entre outros, além de apresentar um alto grau de relevância em relação ao trabalho proposto.

3.2.1 *AspectJ*

AspectJ (KICZALES et al., 2001a) é uma extensão orientada a aspectos da linguagem Java, de propósito geral, desenvolvida pela equipe de pesquisadores do laboratório *Palo Alto Research Center* (PARC), da Xerox. Por se tratar de uma extensão da linguagem Java, todo programa em *AspectJ*, ao ser compilado, é passível de execução em qualquer máquina virtual Java. Para implementar os conceitos da POA, *AspectJ* apresenta novas construções básicas, como: aspectos (*aspects*), pontos de junção (*join points*) e conjunto de pontos de junção (*pointcuts*); e construções específicas, como: declarações intertipos (*inter-type declarations*) e adendos (*advices*).

A principal construção em *AspectJ* é o *aspecto*. Um aspecto define uma funcionalidade específica que pode afetar diferentes partes de um sistema, como, por exemplo, persistência em banco de dados. Além disso, os aspectos podem alterar a estrutura dinâmica de um sistema por meio dos *pontos de junção* ou, ainda, alterar a estrutura estática por meio de uma *declaração intertipos*. Um aspecto normalmente define um *conjunto de pontos de junção* que identifica os pontos de junção e adiciona comportamentos a eles por meio dos *adendos*.

A seguir são apresentadas, detalhadamente, as construções da linguagem *AspectJ*. Para facilitar a compreensão dos novos conceitos definidos pela POA bem como as construções da linguagem *AspectJ*, será utilizado um pequeno sistema, implementado em Java, que trata do estoque de DVD's

[5] Maiores informações sobre as diferentes linguagens orientadas a aspectos podem ser encontradas no *site* da *Aspect-Oriented Software Development Community* (AOSD). URL: `http://aosd.net`

[6] *European Conference for Object-Oriented Programming*. URL: `http://www.ecoop.org`

[7] *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. URL: `http://www.oopsla.org`

(GRADECKI; LESIECKI, 2003). A Figura 7 apresenta o diagrama de classes UML deste sistema.

A classe *"Produto"* contém um atributo e um método relativo ao preço do produto. A classe *"DVD"* representa um produto DVD e estende *"Produto"*. A classe *"Boxset"* também é um produto, e consiste em um conjunto de DVD's relacionados como, por exemplo, a trilogia *Star Wars*, ou O Senhor dos Anéis. Cada classe tem os seus métodos de acesso (leitura e escrita) aos atributos, que são privados. As classes *"DVD"* e *"Boxset"* possuem construtores.

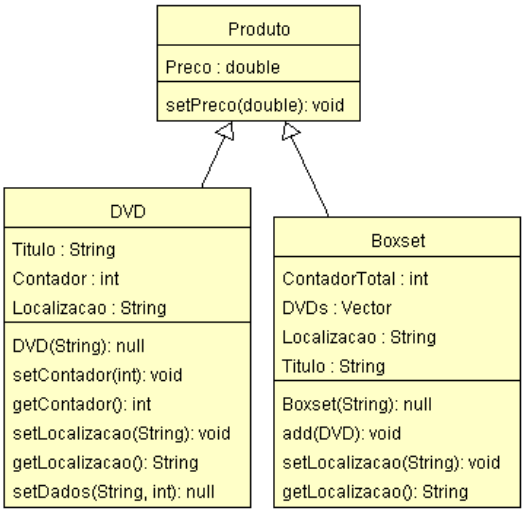


Figura 7: Sistema de Estoque de DVD's

Os atributos *"Contador"* e *"Localizacao"* e os métodos associados a eles nas classes *"DVD"* e *"Boxset"* são usados para localizar o produto, não tendo nenhuma relação fundamental com os produtos DVD e *Boxset*. Eles representam, portanto, um interesse relacionado à manipulação do inventário, que está fora do escopo e sua implementação está espalhada pelo código das classes. Este interesse poderia ser implementado como um interesse primário, estando encapsulado em uma classe, que no caso seria a *"Produto"*. Porém um produto não deve conter informações de inventário. Um produto é uma entidade que contém atributos como preço, título e tamanho, por exemplo (GRADECKI; LESIECKI, 2003).

Os **pontos de junção** (*join points*) em *AspectJ* são pontos bem definidos na execução de uma aplicação. A Figura 8, retirada de (HILSDALE; KICZALES, 2001), mostra um exemplo de fluxo de execução entre dois objetos.

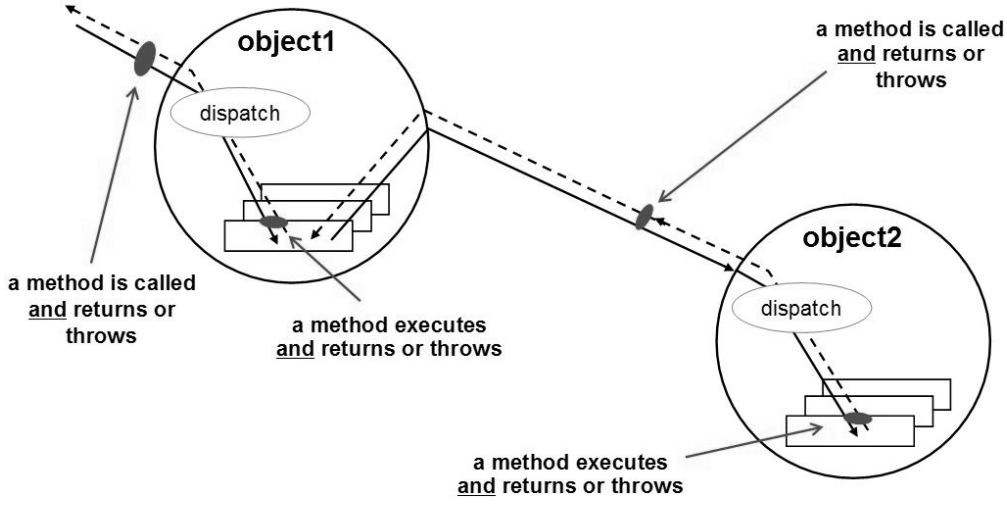


Figura 8: Pontos de junção de um fluxo de execução (HILSDALE; KICZALES, 2001)

O primeiro ponto de junção é a invocação de um método do objeto *"object1"*, o qual pode retornar com sucesso ou lançar uma exceção. O próximo ponto de junção é a execução deste método, que por sua vez também pode retornar com sucesso ou lançar uma exceção. Durante a execução do método do objeto *"object1"*, um outro método, este do objeto *"object2"*, é invocado. Tanto a invocação quanto a execução deste método do objeto *"object2"* são pontos de junção, e, da mesma forma que no objeto *"object1"*, podem retornar com sucesso ou lançar uma exceção.

Os pontos de junção permitem a um aspecto alterar a estrutura dinâmica de um sistema. Alguns exemplos de pontos de junção são: invocação e execução de métodos, execução de construtores, inicialização de classes e de objetos, acesso e modificação de atributos, tratamento de exceções, entre outros. É possível também definir um ponto de junção como sendo um conjunto de vários pontos de junção combinados.

Os pontos de junção são utilizados nas definições dos conjuntos de pontos de junção. Um **conjunto de pontos de junção** (*pointcut*) é uma construção na linguagem *AspectJ*, que visa identificar pontos de junção e obter o contexto da execução desse pontos de junção por meio de critérios bem definidos. Esses critérios podem variar de nomes de funções específicas, ou generalizadas pelo sinal de asterisco (*). Além disso, um conjunto de pontos de junção pode conter operadores lógicos ("&&","||" e "!") e pode ter um nome atribuído para possibilitar a identificação de pontos de junção de diferentes classes.

Um conjunto de pontos de junção é mais do que uma expressão contendo informações sobre os pontos de junção, ele mostra diretamente como um interesse intercepta o fluxo de execução do sistema.

A estrutura de um conjunto de pontos de junção pode ser representada da mesma forma que a assinatura de um método na linguagem Java. A Figura 9 apresenta o trecho da gramática[8] da linguagem *AspectJ* que especifica um conjunto de pontos de junção.

O modificador de um conjunto de pontos de junção **(1)** tanto pode ser *"public"* ou *"private"*, dependendo das considerações no projeto, e indica o escopo do conjunto de pontos de junção no aspecto. Após a definição do tipo do modificador, procede-se a definição do identificador do conjunto de pontos de junção **(2)**. O identificador nada mais é do que um nome, definido por uma seqüência de caracteres e números **(3)** que irá identificar o conjunto de pontos de junção no aspecto, e é análogo ao nome de um método usado tradicionalmente nas classes em Java. Os parâmetros **(4)** são usados para transferir o contexto obtido de um ponto de junção e possuem um tipo Java válido **(5)**. Um parâmetro do conjunto de pontos de junção é seguido de um ou mais designadores

[8]Foi especificada uma gramática da linguagem *AspectJ*, utilizada na abordagem apresentada nesta dissertação, definida com base na literatura (ASPECTJ, 2004; GRADECKI; LESIECKI, 2003; HILSDALE; KICZALES, 2001; KICZALES et al., 2001b)

```
    pointcut     : modifiers* pointcutName '(' parameter_list? ')'
                   (':' ('(')? designator* (')')? )? ';' ;
(1) modifiers    : 'public' | 'private' ('abstract')? ;
(2) pointcutName : Identifier 'pointcut';
(3) identifier   : [a-zA-Z_][0-9a-zA-Z_]* ;
parameter_list   : parameter++',' ; /* lista de parâmetros em Java */
(4) parameters   : Identifier type ;
(5) type         : typeDeclaration ; /* tipos Java válidos */

(6) designator  : designator '||' designator | designator '&&' designator
                | '!' designator | 'call' '(' method_signature ')'
                | 'execution' '(' method_signature ')' | 'get' '(' attrib_signature ')'
                | 'set' '(' attrib_signature ')' | 'handler' '(' type ')'
                | 'initialization' '(' method_signature ')'
                | 'staticinitialization' '(' type ')' | 'within' '(' type ')'
                | 'withincode' '(' method_signature ')' | 'cflow' '(' pointcut ')'
                | 'cflowbelow' '(' pointcut ')' | 'this' '(' arg_list ')'
                | 'target' '(' arg_list ')' | 'args' '(' arg_list ')'
                | 'if' '(' expression ')' | 'adviceexecution' '(' ')'
                | 'preinitialization' '(' ')' | Identifier '(' arg_list ')' ;
```

Figura 9: Sintaxe de um conjunto de pontos de junção em AspectJ

de conjuntos de pontos de junção **(6)**. Estes designadores provêm a definição de como o ponto de junção é utilizado no conjunto de pontos de junção. Existem diversos designadores pré-definidos na linguagem *AspectJ*, denominados designadores primitivos, mas o Engenheiro de Software pode elaborar novos conjuntos de pontos de junção (nomeados ou não) a partir dos já existentes.

Os designadores primitivos dividem-se entre aqueles que selecionam pontos de junção de um determinado tipo, conforme a Tabela 1,

Designador	**Descrição**
call(<*assinatura de método*>)	Quando o método é chamado
execution(<*assinatura de método*>)	Quando o método é executado
get(<*assinatura de atributo*>)	Quando o atributo é acessado
set(<*assinatura de atributo*>)	Quando o atributo é alterado
handler(<*tipo de exceção*>)	Quando a exceção é tratada
initialization(<*assinatura de construtor*>)	Quando o construtor é executado
staticinitialization(<*tipo*>)	Quando a inicialização de classe é executada

Tabela 1: Designadores primitivos

e aqueles que selecionam pontos de junção que satisfazem uma determinada propriedade (por exemplo, em função da classe do objeto no qual o ponto de junção ocorre), conforme a Tabela 2.

Para se utilizar uma combinação dos designadores listados, basta utilizar os operadores lógicos “&&”,“||” e “!” para combiná-los, formando as seguintes construções, conforme a Tabela 3.

Considerando as classes do sistema de estoque de DVD’s, o trecho de código a seguir é a definição de um conjunto de pontos de junção que identifica a invocação do método *“setTitulo”* que é público, sem tipo de retorno (*void*), tem como parâmetro um atributo do tipo *String* e está associado à classe *“DVD”*:

Designador	Descrição
within(<*tipo*>)	Qualquer ponto de junção que ocorra na classe
withincode(<*método*>)	Qualquer ponto de junção que ocorra no método/construtor
cflow(<*conjunto de pontos de junção*>)	Qualquer ponto de junção que ocorra no contexto de um ponto de junção selecionado pelo designador
cflowbelow(<*conjunto de pontos de junção*>)	Idem ao anterior, excluindo os pontos de junção selecionados pelo próprio designador
this(<*tipo*>)	Qualquer ponto de junção que ocorra em um objeto da classe
target(<*tipo*>)	Quando o objeto alvo do **call/get/set** é da classe
args(<*tipo, ...*>)	Quando os argumentos são do tipo especificado
if(<*expressão lógica*>)	Quando a expressão é verdadeira

Tabela 2: Designadores que selecionam pontos de junção que satisfazem uma determinada propriedade

Operador	Descrição
!<*conjunto de pontos de junção*>	Qualquer ponto de junção não selecionado pelo designador
<*conjunto de pontos de junção*> **&&** <*conjunto de pontos de junção*>	Qualquer ponto de junção selecionado por ambos os designadores
<*conjunto de pontos de junção*> **||** <*conjunto de pontos de junção*>	Qualquer ponto de junção selecionado por ao menos um dos designadores

Tabela 3: Operadores lógicos

```
pointcut MudancaTitulo():
  call(public void DVD.setTitulo(String));
```

Os conjuntos de pontos de junção ainda permitem a utilização de coringas (*wildcards*) na especificação dos pontos de junção. Como, por exemplo:

```
pointcut MetodosSet():
  call (public set*(..));
```

seleciona os pontos de junção nos quais quaisquer métodos públicos, que tenham o nome iniciados por *"set"* concatenados com quaisquer outros caracteres e possuam qualquer número de parâmetros, sejam invocados, e

```
pointcut InvocacaoMetodosProduto():
```

```
set(* Produto+.*);
```

seleciona os pontos de junção no qual quaisquer atributos de objetos da classe *"Produto"* ou derivadas sejam alterados.

O Engenheiro de Software pode também criar seus próprios conjuntos de pontos de junção, bem como pode definir designadores de conjunto de pontos de junção, de forma a possibilitar seu uso em mais de um local. A palavra-chave *"pointcut"* é usada para definir um designador de conjunto de pontos de junção (GRADECKI; LESIECKI, 2003).

Um designador de conjunto de pontos de junção pode ainda ser abstrato, quando deve omitir a sua definição. A existência de um designador de conjunto de pontos de junção abstrato obriga o aspecto a ser abstrato. Deve haver então um sub-aspecto concreto que especialize o aspecto abstrato e forneça uma definição para os designadores de conjunto de pontos de junção abstratos. Tanto designadores de conjunto de pontos de junção primitivos como os de usuário podem ser usados na definição dos conjuntos de pontos de junção do usuário.

O complemento dos conjuntos de pontos de junção são os **adendos**. Um adendo é o código que é executado quando um ponto de junção é identificado por um conjunto de pontos de junção, ou seja, eles definem quando (antes, no momento ou após) e como o aspecto irá se comportar no ponto de junção.

A Figura 10 apresenta como os adendos são definidos, de acordo com a gramática da linguagem *AspectJ*.

```
advice_declaration: type? adviceType '(' parameter_list? ')' ('[]')*
                    throw_part? afterQualifier? ':'
                    (Identifier '(' arg_list? ')' | designator*) '{'
                    adviceBody* '}' ;
(1) adviceType    : 'before' | 'after' | returnType 'around' ;
(2) returnType    : typeOrPrimitive ;
(3) afterQualifier: 'throwing' ('(' parameter_list ')')?
                  | 'returning' ('(' parameter_list ')')? ;
(4) parameter_list: parameter++',' ; /* lista de parâmetros em Java */
(5) adviceBody    : statement ; /* corpo de método em Java */
    designator    : ... /* definido na sub-seção anterior */
```

Figura 10: Sintaxe de um adendo em *AspectJ*

Os adendos podem ser de três tipos **(1)**: "anterior" (*"before"*), que executa quando a computação alcança a invocação do método e **antes** de sua execução; "posterior" (*"after"*), que executa **após** a execução do corpo do método e antes do controle retornar ao cliente; e "de contorno" (*"around"*), que executa **quando alcança** o ponto de junção e possui controle explícito sobre a execução (ou não) da computação originalmente associada ao ponto de junção.

Os adendos de contorno são os que têm uma maior influência sobre o código do sistema, já que eles substituem o comportamento original do ponto de junção. Além disso, o adendo de contorno é o único que possui um tipo de retorno **(2)**.

Os adendos posteriores que possuem o qualificador *"returning"* **(3)**, são executados somente quando o ponto de junção retorna normalmente, sem erros, o controle ao aspecto. Já os que possuem o qualificador *"throwing"* **(3)** são executados somente quando o ponto de junção é finalizado lançando uma exceção.

Um adendo também pode declarar parâmetros **(4)**, que devem ser correspondidos na declaração dos conjuntos de pontos de junção. Esses parâmetros ficam disponíveis na execução do corpo do adendo **(5)**.

Considerando um conjunto de pontos de junção responsável por identificar a invocação do construtor da classe *"DVD"*, na Figura 11 a seguir tem-se um exemplo de um adendo anterior.

```
pointcut Dados(Object obj):
  execution (public void DVD.setDados(..)) && this(obj);

pointcut Dados2(Object obj):
  call (public void DVD.setDados(..)) && this(obj);

pointcut Todos(Object obj):
  Dados(obj) || Dados2(obj);

before(Object obj): Todos(obj) {
                          (1)          (2)
  System.out.println(thisJoinPoint.toLongString());
  if(obj instanceof DVD)
    System.out.println("Objeto DVD");
  else if(obj instanceof Boxset)
    System.out.println("Objeto Boxset");
  else
    System.out.println("Objeto desconhecido");
}
```

Figura 11: Exemplo de um adendo anterior

O atributo *"thisJoinPoint"* **(1)** é uma variável usada para acessar o contexto do ponto de junção. O método *"toLongString()"* **(2)** retorna uma *"String"* com uma representação por extenso do ponto de junção. O resultado da execução do adendo anterior do conjunto de pontos de junção *"Todos(obj)"*, usando o exemplo da classe *"DVD"* é:

```
execution (public DVD(java.lang.String))
Objeto DVD
```

O resultado mostra que o construtor da classe *"DVD"* foi executado. Quando a identificação foi determinada, o código do adendo do conjunto de pontos de junção foi executado e exibiu a

representação por extenso do ponto de junção identificado.

Na identificação de um ponto de junção, por um conjunto de pontos de junção, todos os adendos são executados antes e/ou após o ponto de junção. A execução desses adendos é definida com base em diversos fatores, como, por exemplo, o tipo do adendo, a especificidade do conjunto de pontos de junção e regras de precedência explícita entre os aspectos que contém os conjuntos de pontos de junção, por meio da utilização da construção *"declare precedence"*. Por exemplo:

```
aspect AspectoExcecao declare precedence AspectoLog { ... }
```

Assim, nos pontos de junção comuns aos dois aspectos, os adendos do aspecto *"AspectoExcecao"* tem prioridade na execução em relação aos do aspecto *"AspectoLog"*.

Até o momento, todas as construções apresentadas são de caráter dinâmico, pois elas alteram a estrutura do sistema por meio da interceptação de pontos bem definidos no fluxo de execução (pontos de junção) e da adição de comportamento antes e/ou depois dos mesmos, ou ainda, obtendo o controle total da execução. Porém, a linguagem *AspectJ* também possui construções, denominadas declarações intertipos, para alterar estaticamente a estrutura do sistema, adicionando membros (atributos, métodos e construtores) a uma classe, alterando a hierarquia de herança das classes e interfaces do sistema (GRADECKI; LESIECKI, 2003).

As **declarações intertipos** (*inter-type declarations*) são consideradas interesses transversais estáticos porque não dependem da execução. O restante do sistema trata a nova estrutura como se ela fosse declarada naturalmente, independentemente de estar sujeito a novas regras de escopo. Embora as declarações intertipos possam proporcionar aos aspectos a habilidade para, "exteriormente", alterar as classes, o resultado final é combinado perfeitamente no código Java durante o processo de combinação aspectual.

A Figura 12 apresenta um exemplo que ilustra diferentes formas de utilização de declarações intertipos em AspectJ:

Em **(1)** um atributo do tipo *"String"* denominado *"Ano"* está sendo introduzido na classe *"Produto"*. Já em **(2)** é declarada uma *interface* denominada *"IProduto"* que possui a assinatura de dois métodos: *"void setAno(String inAno);"* e *"String getAno();"*. Em **(3)**, é declarado que a classe *"Produto"* implementará a *interface "IProduto"*. Finalmente, em **(4)**, está a declaração da implementação dos corpos dos métodos da *interface "IProduto"*.

A partir do exemplo da Figura 12, pode-se notar que atributos definidos como uma declaração intertipos na classe, ficam disponíveis para uso pelos métodos também assim declarados. A *interface* adicionada pode ser usada em designadores de conjuntos de pontos de junção para selecionar

```
(1) public String Produto.Ano;

(2) interface IProduto {
      void setAno(String inAno);
      String getAno();
    }

(3) declare parents: Produto implements IProduto;

(4) public String IProduto+.getAno(){
      return this.Ano;
    }
    public void IProduto+.setAno(String inAno){
      this.Ano = inAno;
    }
```

Figura 12: Exemplo de declaração intertipos em *AspectJ*

a classe na qual a *interface* foi implementada.

A principal construção em *AspectJ* é o **aspecto** (*aspect*). Cada aspecto define uma funcionalidade específica que pode afetar várias partes de um sistema, como, por exemplo, tratamento de exceções. Um aspecto é uma construção que encapsula pontos de junção, conjuntos de pontos de junção, declarações intertipos e código de adendos. Um aspecto, assim como uma classe Java, pode definir membros (atributos e métodos) e uma hierarquia de aspectos, por meio da definição de aspectos especializados. O formato de um aspecto é apresentado na Figura 13.

```
    aspect           : modifier* ('privileged')? ('static')? 'aspect'
                       Identifier (classIdentifier)? (instatiation)? ;
(1) modifiers        : ('public' | 'private') ('abstract')? ;
(2) Identifier       : [a-zA-Z_][0-9a-zA-Z_]* ;
(3) classIdentifier  : ('dominates' name)? ('extends' name)? ;
(4) instantiation    : 'issingleton' | 'perthis' '(' designator* ')'
                     | 'pertarget' '(' designator* ')'
                     | 'percflow' '(' designator* ')'
                     | 'percflowbelow' '(' designator* ')' ;
```

Figura 13: Sintaxe de um aspecto em *AspectJ*

O modificador **(1)** mais comumente utilizado nos aspectos é o *"public"*, porém, algumas vezes, um aspecto pode ser declarado como *"abstract"*, que pode ser utilizado para implementar o reuso de aspectos (PIVETA; ZANCANELLA, 2003). O modificador *"private"* é o menos utilizado. Os modificadores especificados para os aspectos seguem as mesmas regras para as classes Java. O modificador *"privileged"* permite que o aspecto tenha acesso interno aos atributos privados da classe.

Um aspecto, assim como uma classe Java, possui um nome identificador **(2)**. Ele pode ainda implementar uma hierarquia de herança, por meio da palavra-chave *"extends"* **(3)**, sendo que o aspecto pode derivar de outro aspecto ou de uma classe. Apenas os aspectos abstratos (com

conjuntos de pontos de junção abstratos) podem ser especializados.

Por padrão, existe apenas uma instância para cada aspecto, já que a declaração de construtores públicos não é disponível. Porém, é possível definir uma associação por instância de objeto ou por ponto de junção selecionados por um designador, usando os modificadores de instanciação **(4)**: *"issingleton"*, *"perthis"*, *"pertarget"*, *"percflow"* e *"percflowbelow"*.

O *"issingleton"* é o tipo padrão atribuído para todos os aspectos que não possuem aspectos pais ou uma declaração específica de um tipo de instanciação. Ele pode ser atribuído a um aspecto para assegurar que somente um objeto do tipo aspecto foi criado no sistema. A estrutura sintática é como se segue:

```
aspect ID issingleton { ... }
```

Já o *"perthis"* e o *"pertarget"* criam uma instância do aspecto para o objeto que for o corrente no momento de execução de um ponto de junção previamente especificado.

```
aspect ID perthis (conjunto de pontos de junção){ ... }
aspect ID pertarget (conjunto de pontos de junção) { ... }
```

Quando os designadores *"cflow"* e o *"cflowbelow"* são utilizados em um aspecto, eles identificam um fluxo de execução com base nos pontos de junção associados aos designadores. O *"percflow"* e o *"percflowbelow"* podem ser utilizados para forçar a instanciação de um objeto do tipo aspecto toda vez que o fluxo de execução for iniciado.

```
aspect ID percflow (conjunto de pontos de junção){ ... }
aspect ID percflowbelow (conjunto de pontos de junção) { ... }
```

Um exemplo de um aspecto para o tratamento de exceção, considerando a classe *"DVDException"*, pode ser implementado como se segue, na Figura 14.

Neste conjunto de pontos de junção e no código do adendo, toda exceção lançada pela classe será capturada pelo aspecto *"AspectoExcecaoDVD"*, por causa do tipo da classe *"Exception+"* usada no designador *"handler"* **(1)**. Quando uma exceção de um objeto é passada para o código do adendo, a palavra-chave *"instanceof"* **(2)** determina se a classe do ponto de junção é ou não a *"DVDException"*. Se a classe for a *"DVDException"*, o nome atribuído ao objeto é exibido **(3)**.

```
  public aspect AspectoExcecaoDVD {

    pointcut handle(Exception e):
(1)   handler(Exception+) && args(e);
    before : Exception (e): handle (e) {
(2)   if (e instanceof DVDException) {
(3)     System.out.println(((DVDException) e).getTitulo());
      }
      System.out.println(thisJoinPoint.toLongString());
  }
```

Figura 14: Exemplo de um aspecto de exceção em *AspectJ*

3.3 Projeto de Software Orientado a Aspectos

Em um processo de desenvolvimento de software, diversas atividades estão envolvidas, desde a especificação dos requisitos, análise, projeto, até a implementação em um código executável.

O processo de Desenvolvimento de Software Orientado a Aspectos (DSOA) organiza o projeto orientado a aspectos e possibilita a identificação de aspectos em um alto nível de abstração nas primeiras fases do processo, favorecendo assim, a aprendizagem e a documentação do projeto de aspectos de uma forma mais intuitiva. A partir de então, esta documentação e o conhecimento adquirido no processo, facilitam e estimulam o reuso de aspectos (SUZUKI; YAMAMOTO, 1999).

Os conceitos básicos do paradigma orientado a aspectos, como pontos de junção, conjuntos de pontos de junção, entrelaçamento, adendos e aspectos são fundamentos que devem estar presentes em todo o processo de desenvolvimento de software orientado a aspectos, desde a especificação até a implementação. Porém, sabe-se que estes conceitos foram originalmente propostos considerando-se apenas a sua implementação, por meio de uma linguagem de programação (KICZALES et al., 2001b). Contudo, a definição de um padrão para o projeto de software orientado a aspectos, a partir de modelos e técnicas de modelagem orientadas a aspectos, independentes da linguagem de programação, vem despertando o interesse de pesquisadores (BANIASSAD; CLARKE, 2004), (CLARKE; WALKER, 2002), (CLEMENTE; SÁNCHEZ; PÉREZ, 2002), (CHAVEZ, 2004), (HERRERO et al., 2000), (PAWLAK et al., 2002) e (STEIN; HANENBERG; UNLAND, 2002). Todavia, ainda é uma área em que a pesquisa foi pouco desenvolvida.

Os pesquisadores dessa área buscam, entre outras coisas, definir uma linguagem de modelagem na qual um modelo mais completo para DSOA irá se basear. Segundo Clarke e Walker (2002) uma linguagem de modelagem que apóie o projeto de software orientado a aspectos deve atender a alguns objetivos principais, a saber:

- **Independência da linguagem de implementação:** uma linguagem de modelagem orien-

tada a aspectos não pode estar baseada em uma linguagem de programação orientada a aspectos específica, deve ser genérica;

- **Composição em nível de projeto:** a composição de elementos em nível de projeto deve ser possível, para possibilitar ao Engenheiro de Software testar a composição desses elementos antes da implementação, ou unir no mesmo projeto elementos orientados a aspectos e não orientados a aspectos, como, por exemplo, aspectos e componentes de negócios;
- **Compatibilidade com as linguagens existentes:** uma linguagem de modelagem orientada a aspectos deveria estar baseada em uma linguagem de modelagem já existente, como, por exemplo, a UML, para minimizar o impacto na mudança das técnicas antigas para as novas.

Neste contexto, Chavez e Lucena (CHAVEZ; LUCENA, 2001; CHAVEZ, 2004) propõem um modelo de projeto para o DSOA que incorpora as principais características da POA, e é independente de linguagem e de processo. Por meio do modelo de projeto proposto, que é uma extensão do modelo orientado a objetos, os autores pretendem reduzir o tempo gasto no DSOA e a distância existente entre o projeto e a implementação no qual linguagens orientadas a aspectos são utilizadas.

O modelo proposto compreende de três facetas: **uma faceta estrutural**, que objetiva a modelagem da estrutura dos aspectos e dos componentes e de como eles estão relacionados em relação à interceptação do fluxo de execução do sistema e nos mecanismos de composição; **uma faceta dinâmica**, que objetiva modelar as interações entre as instâncias dos aspectos e as instâncias dos componentes que eles interceptam, por meio da perspectiva do aspecto; e por fim uma **faceta composicional**, que objetiva modelar o processo de composição (*weaving*) para facilitar o entendimento dos efeitos causados pela combinação entre aspectos e componentes em um alto nível de abstração.

3.3.1 Abordagens para o projeto orientado a aspectos

A necessidade de uma notação satisfatória para o projeto de software orientado a aspectos foi reconhecida anos após o paradigma ter sido criado. Dentre as diferentes propostas para estender a UML, destaca-se a proposta de Suzuki e Yammamoto (SUZUKI; YAMAMOTO, 1999) e a de Pawlak e outros (PAWLAK et al., 2002).

3.3.1.1 Suzuki e Yammamoto

A primeira proposta de extensão da UML com conceitos para o projeto de software orientado a aspectos foi apresentada por Suzuki e Yammamoto (SUZUKI; YAMAMOTO, 1999). Nesta

abordagem, uma nova meta-classe UML chamada *"aspect"* é introduzida. Esta meta-classe está relacionada à classe base por meio de um relacionamento de realização. Esta proposta tem dois problemas.

Primeiro, os autores apresentam somente uma notação que pode ser usada para representar o conceito de declarações intertipos. Não está claro como os conjuntos de pontos de junção e os adendos podem ser especificados na UML e como será representada a forma que os relacionamentos de entrecorte (*crosscutts*) afetam o comportamento das classes.

Segundo, o uso de um relacionamento de realização para modelar a relação entre um aspecto e as classes que ele entrecorta não está totalmente de acordo com a semântica da *AspectJ*. Na UML, *"uma realização é um relacionamento entre um elemento de modelo específico e o elemento de modelo que o implementa"* (UML, 2004). Em *AspectJ*, entretanto, um adendo não é uma pura declaração de um entrelaçamento. Em *AspectJ*, um adendo declara e implementa as características de um entrelaçamento.

A Figura 15 apresenta a notação de Suzuki e Yammamoto. Um aspecto é representado por um retangulo com o esteriótipo <<aspect>>.

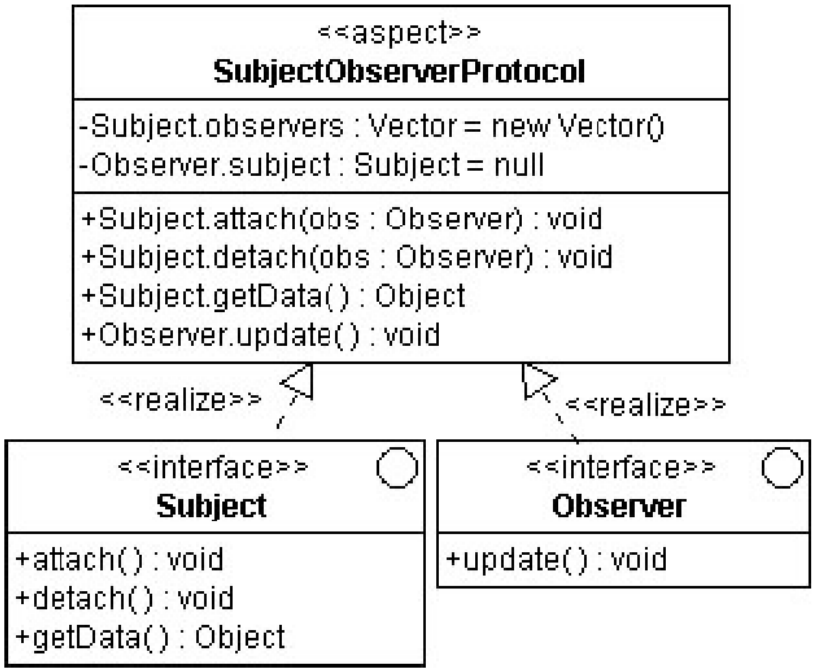


Figura 15: Notação de Suzuki e Yammamoto

3.3.1.2 Pawlak e outros

A proposta de Pawlak e outros (PAWLAK et al., 2002) para o Projeto de Software Orientado a Aspectos, por meio da extensão da UML, é a que se apresenta mais interessante. São definidos três

novos conceitos na UML: (1) grupos (*groups*), que provêm meios de classificação para entidades distribuídas e heterogêneas, (2) relações de conjunto de pontos de junção (*pointcut relations*), que permitem ao Engenheiro de Software definir entrecortes (*crosscutts*) dentro do programa funcional, e (3) a meta-classe aspecto (*aspect-classes*), que implementa a extensão do programa nos pontos de entrecorte (*crosscutting points*) especificados pelas relações de conjuntos de pontos de junção.

Porém, alguns pontos sobre o trabalho de Pawlak e outros apresentam problemas: a utilização do papel de um relacionamento para representar os pontos de junção acaba poluindo visualmente o modelo, visto que a definição de pontos de junção normalmente envolve um grande número de caracteres. Outro ponto é a utilização de caracteres especiais como sinal de exclamação ("!") que pode possuir outro significado em um linguagens de programação, resultando em confusão.

A Figura 16 mostra os elementos utilizados por essa notação. O estereótipo <<aspect>> é utilizado para representar um aspecto. O estereótipo <<pointcut>> denota uma relação unidirecional do aspecto para a classe C, indicando que esse aspecto intercepta e interrompe a execução de um método da classe em algum ponto relevante da execução e executa um algum código antes ou depois da execução da classe. O papel *"r1"* é o nome de um método do aspecto definido no aspecto A que afeta algum ponto dos componentes, definido pelo papel *"r2"*. O rótulo *"r2"* é definido como um conjunto de pontos de junção no formato *T[expressão]*, onde *T* pertence ao conjunto descrito na Tabela 2 e a expressão representa algum ponto no programa como, por exemplo, a assinatura de um determinado método, o nome de uma classe ou mesmo alguns dos caracteres especiais utilizados para representar palavras-chaves da notação. As cardinalidades *"card1"* e *"card2"* indicam o número de instâncias, tanto dos aspectos quanto das classes e o *"tag"* é uma etiqueta valorada para estender a semântica do relacionamento.

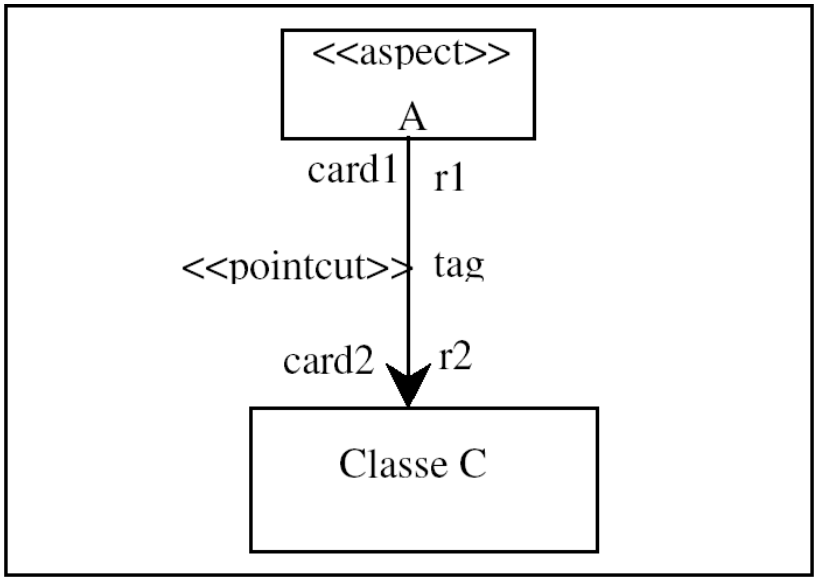


Figura 16: Notação para Programas Orientados a Aspectos de Pawlak e outros

3.3.1.3 Stein e outros

Stein e outros também propuseram uma extensão da UML para a modelagem de programas orientados a aspectos implementados em *AspectJ* (STEIN; HANENBERG; UNLAND, 2002). A extensão proposta é um modelo de projeto e denomina-se *AODM* (*Aspect Oriented Design Model*). Este modelo foi proposto a partir da linguagem de modelagem UML, adicionando os conceitos clássicos da programação orientada a aspectos. Além disso, também reproduz o mecanismo de composição (*weaving*). Os aspectos são representados como classes UML com estereótipo <<*aspect*>> e podem participar em relacionamentos de associação, generalização e dependência. Eles contém elementos de entrecorte (*crosscutts*) que podem alterar a estrutura e o comportamento de um modelo. A estrutura é alterada por meio de *templates* de colaboração (*collaboration templates*[9]), que possuem o estereótipo <<*introduction*>> e o comportamento por meio de operações com estereótipo <<*advice*>>. Os aspectos também possuem etiquetas valoradas que especificam como será a sua instanciação.

Enquanto os *templates* de colaboração descrevem relacionamentos estruturais, os comportamentais entrecortam as ligações (*links*[10]) utilizados para relacionar duas ou mais classes e são representados com diagramas de interação da UML. O *AODM* também possui estereótipos que permitem interceptar invocações dentro de uma mesma entidade, tais como: construtores, tratamento de exceções, inicializadores de classes e objetos de acesso a campos.

Embora essa notação seja bastante completa, ela não influenciou a notação proposta por este documento devido à dificuldade de leitura que ela apresenta e também à falta de apoio automatizado.

A Figura 17 mostra apenas a representação gráfica de um aspecto com o estereótipo <<aspect>> denominado *Timing*. Nota-se que os autores definiram estereótipos tanto para pontos de corte (<<pointcuts>>) quanto para sugestões (<<advice>>). Além disso, eles também utilizam os *"templates de colaboração"* que são representados pelas elipses tracejadas na parte inferior do aspecto. Esses *templates* são utilizados com o estereótipo <<introduction>>.

3.4 Considerações Finais

Este capítulo apresentou uma visão geral da programação orientada a aspectos, discutindo de forma detalhada os seus objetivos e conceitos, bem como a aplicação destes no DSOA. Também

[9] Um *template* é um elemento de modelo parametrizado utilizado para gerar outros elementos do modelo através de passagem de parâmetro.

[10] Correspondem aos pontos de junção em *AspectJ*.

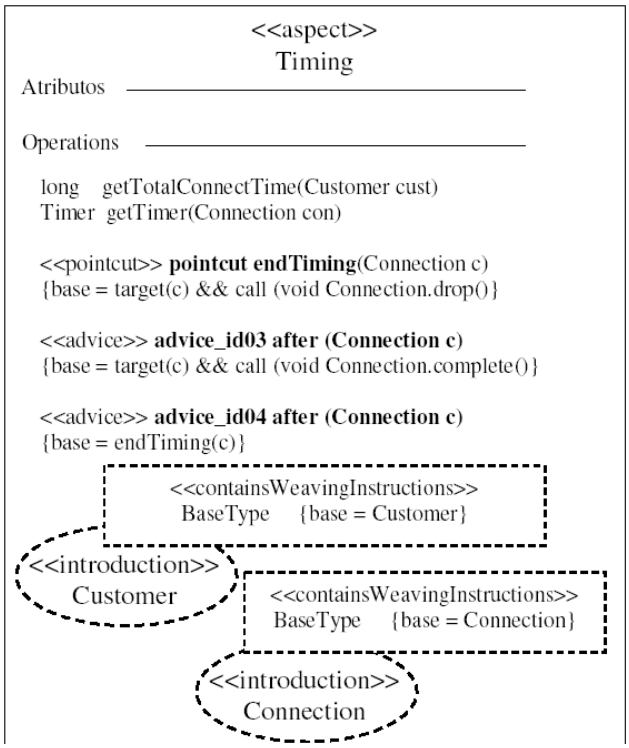


Figura 17: Notação de Aspecto em AODM

foram introduzidas a especificação de uma linguagem orientada a aspectos, descrevendo os seus componentes básicos. A linguagem *AspectJ* foi detalhada, descrevendo seus construtores básicos por meio de exemplos práticos. Além disso, foram relatados os principais tópicos de pesquisa relativos à definição de uma padronização no projeto de software orientado a aspectos, destacando as principais pesquisas que vêm sendo desenvolvidas.

4 *Phoenix: Uma Abordagem para Reengenharia de Software Orientada a Aspectos*

Nos capítulos anteriores foram apresentados os principais conceitos envolvidos nesta pesquisa, que forneceram o embasamento teórico para o desenvolvimento de uma abordagem com o objetivo de auxiliar a migração de sistemas orientados a objetos para aspectos, por meio da Reengenharia de Software. Neste contexto, neste capítulo apresenta-se em detalhes a abordagem *Phoenix* para a Reengenharia de Software Orientada a Aspectos. A abordagem baseia-se em técnicas de Engenharia Reversa e Reengenharia de Software e é apoiada por diversos mecanismos que auxiliam o Engenheiro de Software no processo, automatizando grande parte das atividades da Reengenharia.

4.1 Mecanismos da Abordagem proposta

A abordagem *Phoenix* combina diferentes técnicas e mecanismos para a Reengenharia de Software, com base na experiência do grupo no qual o aluno e seu orientador fazem parte (ALVARO et al., 2003; GARCIA et al., 2004c, 2005; PRADO et al., 2004).

4.1.1 Desenvolvimento de Software Orientado a Aspectos

Como visto no Capítulo 3, muitas funções existentes nos sistemas de software são inerentemente difíceis de decompor e isolar, reduzindo a legibilidade e a manutenibilidade desses sistemas.

O Desenvolvimento de Software Orientado a Aspectos (DSOA) surgiu como um paradigma para separar os interesses transversais por meio de técnicas de geração de código que combinam aspectos dentro da lógica do sistema (KICZALES et al., 1997).

A *Separação de Interesses* é um princípio bem definido na Engenharia de Software. Um interesse é parte do problema que deve ser tratado como uma única unidade conceitual (TARR et al., 1999). Interesses são modularizados por meio do desenvolvimento de software utilizando

diferentes abstrações, providas por linguagens, métodos e ferramentas.

Na Abordagem *Phoenix*, o DSOA é utilizado para aumentar a reusabilidade do conhecimento extraído, pelo encapsulamento dos interesses funcionais e não-funcionais em unidades separadas. Entretanto, antes desse encapsulamento, é necessário ter uma forma de identificar, no código do sistema, onde estes diferentes interesses estão localizados. Para isso, utilizam-se técnicas de Mineração de Aspectos.

4.1.2 Mineração de Aspectos

Na literatura pode-se encontrar diversas propostas (DEURSEN; MARIN; MOONEN, 2003; JANZEN; De Volder, 2003; IWAMOTO; ZHAO, 2003) para dar suporte à identificação de aspectos em estágios iniciais do processo de desenvolvimento de software. O principal objetivo de migrar sistemas orientados a objetos para aspectos é melhorar o entendimento do sistema, aumentando assim a manutenibilidade e dando uma maior garantia para a sua evolução.

Várias preocupações relativas ao uso do DSOA são levantadas como, por exemplo, o risco de obter um código mais entrelaçado, conhecido como *"código espaguete"*, pela não utilização correta dos conceitos da Orientação a Aspectos e o uso indiscriminado de abordagens *ad-hoc* para o projeto de sistemas, pelo fato de não haver um padrão definido para expressar a modelagem de sistemas orientados a aspectos. Essas preocupações levantam a seguinte questão: *Quando o DSOA é necessário e quando somente a Programação Orientada a Objetos resolve o problema ?*

Neste contexto, a mineração de aspectos pode ajudar a resolver esta questão. Técnicas de mineração de software ajudam a encontrar informações valiosas no código de um sistema, tornando estas informações disponíveis aos Engenheiros de Software envolvidos na evolução daquele sistema. Um bom exemplo de mineração de software é a *extração de regras de negócio*.

A mineração de software é apoiada por técnicas de exploração de software (MOONEN, 2003). Ela envolve três passos: (1) coletar dados do código fonte, (2) inferir conhecimento com base na abstração dos dados coletados, e (3) apresentar a informação usando, por exemplo, hipertexto ou outros recursos para visualização.

A identificação dos candidatos a aspectos requer primeiramente uma clara idéia de quais aspectos se pretende encontrar. Assim, um estudo sobre os aspectos genéricos ou específicos de um domínio é um pré-requisito para a mineração de aspectos. Por exemplo, antes de minerar o interesse de persistência em banco de dados, é necessário entender como este interesse é tratado, e como ele é tratado particularmente no sistema em questão. Esse estudo pode envolver a análise da documentação disponível e a busca pela chamada de métodos específicos, chamadas a constru-

tores, acesso a atributos e outros pontos de execução bem definidos, visando descobrir possíveis pontos de junção envolvendo o interesse a ser minerado.

Para realizar a mineração de aspectos existe a necessidade de se fazer a análise (*parsing*) dos programas orientados a objetos e verificar os locais nos quais existem códigos duplicados, difusos ou referentes a diversos interesses de projeto.

Diversos trabalhos focam na mineração de aspectos (HANNEMANN; KICZALES, 2001; ROBILLARD; MURPHY, 2002) para auxiliar na identificação de interesses transversais embutidos no código do sistema. Existem duas abordagens principais:

- **Mineração baseada em texto:** A mineração de dados baseada em texto procura por padrões no código fonte, utilizando como base os nomes de classes, métodos e atributos. Ela é interessante caso sejam utilizadas convenções de nomeação para os elementos do sistema. A não utilização de um padrão para tal pode dificultar no processo de identificação de interesses transversais.

- **Mineração baseada em tipos:** A mineração por meio de tipos procura pela ocorrência dos diversos tipos (i.e. classes) definidos no aplicativo a ser analisado. Ela permite encontrar pontos em um sistema nos quais o acoplamento e a coesão possam deixar a desejar. Um único módulo que utiliza alguns poucos tipos é candidato a possuir uma coesão alta e baixo acoplamento, por exemplo. A mineração baseada em tipos funciona melhor do que a mineração baseada em texto quando não existem convenções de nomeação. Ela não é muito interessante quando instâncias de um mesmo tipo são utilizadas para diferentes propósitos.

O ideal é que essas duas abordagens sejam utilizadas em conjunto, já que uma complementa a outra.

Uma vez identificados, os interesses devem ser extraídos e encapsulados em aspectos. As transformações requeridas para a geração de aspectos usando os resultados da mineração podem ser consideradas similares a refatorações (*refactorings*) (FOWLER et al., 1999).

4.1.3 Refatorações Orientadas a Aspectos

Após a extração dos interesses transversais, a extração deve ser planejada por meio de técnicas bem definidas. O problema é como extrair os interesses que estão entrelaçados dentro do código da aplicação e espalhados por diversas classes. Nesta pesquisa utilizou-se para isso a POA (KICZALES et al., 1997), que auxilia na separação desses diferentes interesses em unidades separadas. Para extrair estes interesses dos sistemas orientados a objetos em aspectos, é possível utilizar refatorações

(FOWLER et al., 1999), adaptadas ao paradigma orientado a aspectos.

Refatoração é uma técnica utilizada para reestruturar código orientado a objetos de uma forma disciplinada. A intenção dela é aprimorar a legibilidade e a compreensão do código orientado a objetos. A maioria das técnicas de refatoração se propõe a aumentar a modularidade e eliminar a redundância do código. Considerando que as mesmas vantagens são obtidas por meio do DSOA, parece ser natural aplicá-los no mesmo processo de desenvolvimento (DEURSEN; MARIN; MOONEN, 2003). O benefício de se utilizar ambas as abordagens é considerável. Refatoração pode ajudar a reestruturar o código orientado a objetos segundo o paradigma orientado a aspectos, tornando viável a migração de sistemas orientados a objetos para orientados a aspectos.

A partir da pesquisa de catálogos de refatorações existentes como, por exemplo (FOWLER et al., 1999), foram identificadas diversas similaridades entre refatorações para melhoria de código e a extração de aspectos. Assim, foi definido um catálogo[1] de refatorações (GARCIA et al., 2004), com base em refatorações existentes (FOWLER et al., 1999; HANENBERG; OBERSCHULTE; UNLAND, 2003; IWAMOTO; ZHAO, 2003), que são utilizadas para extrair os aspectos do código fonte orientado a objetos. A Figura 18 mostra a interação entre essas refatorações orientadas a aspectos especificadas, onde as refatorações sombreadas (em cinza) não são especificadas nesta dissertação.

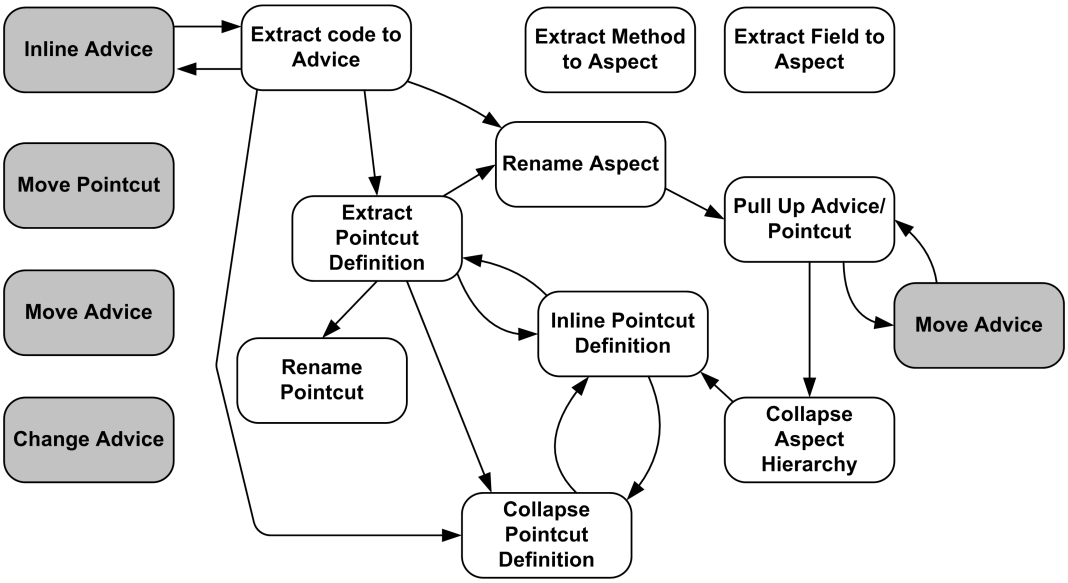


Figura 18: Refatorações para extrair interesses transversais

O catálogo definido contém refatorações para extrair os interesses de acordo com os principais conceitos do DSOA, como, por exemplo, a extração de métodos e atributos de classes para aspectos, definição de conjuntos de pontos de junção, construção de adendos, entre outros.

As refatorações apresentadas aqui são propostas a partir da experiência do aluno e seu orien-

[1]Refatorações são sistematicamente organizadas em catálogos, de um modo análogo aos padrões de projeto (GAMMA et al., 1995).

tador em DSOA e na realização de estudos de caso. É importante ressaltar que essas refatorações não abrangem todos os problemas e nem pretendem ser a solução completa para a extração de interesses transversais do código orientado a objetos, mas representam o primeiro passo para a criação de um catálogo com este objetivo.

Após a extração dos aspectos, obter uma representação em alto nível de abstração facilitaria o entendimento do sistema para futuras atividades de manutenção, a evolução do software, bem como estimularia o reuso. A abordagem *Phoenix* utiliza transformações de software para auxiliar na obtenção dessa representação.

4.1.4 Transformação de Software

A abordagem *Phoenix* utiliza um sistema transformacional para auxiliar e semi-automatizar a tarefa de obtenção de uma representação em alto nível dos elementos recuperados.

Diferentes Sistemas Transformacionais (BAXTER; PIDGEON, 1997; BOYLE, 1989; CASTOR et al., 2001; CORDY; CARMICHAEL, 1993; JACOBSON; LINDSTROM, 1991; REFINE, 1992; RESCUEWARE, 2004; WILE, 1993) têm sido utilizados para automatizar as atividades relativas à Reengenharia de Software, destacando-se o Draco-PUC, que vem sendo utilizado em diversos projetos de pesquisa como, por exemplo, (ABRAHÃO; PRADO, 1999; BERGMANN; LEITE, 2000, 2002; LEITE; SANT'ANNA; FREITAS, 1994; LEITE; SANT'ANNA; PRADO, 1996; PRADO, 1992; PRADO et al., 2004) e implementa as idéias de transformação de software orientada a domínios (NEIGHBORS, 1980, 1983, 1989).

O Draco-PUC é uma ferramenta baseada nas idéias do paradigma Draco (NEIGHBORS, 1983), implementando a grande maioria das características importantes em um sistema transformacional de software. Dentre essas características, destaca-se um sistema de *parser* de grande expressividade, baseado no *Bison* (DONNELLY; STALLMAN, 2004), aliado a outros recursos, como, por exemplo, o *backtracking* (FREITAS; LEITE; SANT'ANNA, 1996). As transformações no Draco-PUC são escritas com base na sintaxe das linguagens dos domínios cujos programas se deseja transformar e na sintaxe da linguagem do sistema de transformação denominado TFMGEN (PRADO, 1992) do próprio Draco-PUC.

Além disso, o Draco-PUC possui um código portátil e que pode operar em diferentes versões de sistemas operacionais, como *Unix*, *Windows 9x*, *NT*, *2000* e *XP*.

Diversas pesquisas foram desenvolvidas para comprovar a empregabilidade do Draco-PUC na transformação de software de diferentes domínios, conforme comprovam os resultados apresentados em (ALVARO et al., 2003; BOSSONARO, 2004; FUKUDA, 2000; JESUS; FUKUDA; PRADO, 1999;

MORAES, 2004; NOGUEIRA, 2002; NOVAIS, 2002). Estas pesquisas também serviram para testar o Draco-PUC e identificar as dificuldades na construção dos domínios. Ao longo destas pesquisas, grande parte das dificuldades foram superadas e os erros encontrados foram corrigidos, tornando o Draco-PUC mais estável e confiável.

Assim, considerando: as experiências realizadas com o Draco-PUC no DC/UFSCar e em outras instituições de pesquisa e dada a versatilidade e capacidade do Draco-PUC de apoiar diferentes tarefas da Reengenharia de Software, decidiu-se utilizar o Draco-PUC para auxiliar na recuperação do projeto do sistema, com características da Orientação a Aspectos.

4.1.5 Ferramenta *CASE*

Outro tipo de ferramenta freqüentemente utilizada na Engenharia de Software são as ferramentas CASE de modelagem, auxiliando tanto no projeto quanto no reprojeto de sistemas.

Dentre as diferentes ferramentas CASE (ARGOUML, 2003; TOGETHER, 2004; RATIONAL, 2004; OMONDO, 2004) destaca-se a MVCASE[2] (ALMEIDA et al., 2002a, 2002b). A MVCASE é uma ferramenta orientada a objetos que dá suporte à especificação textual e gráfica de requisitos do sistema, em diferentes níveis de abstração, utilizando a notação UML (BOOCH; RUMBAUGH; JACOBSON, 1999).

Outra característica da MVCASE é o suporte para a geração de código em uma linguagem de programação orientada a objetos como, por exemplo, Java, a partir de especificações em alto nível de abstração. Outro motivo da escolha da MVCASE é devido à mesma ser uma ferramenta desenvolvida em Java e, portanto, multi-plataforma e de livre distribuição, assim como o Draco-PUC.

Embora a MVCASE dê suporte a diferentes recursos de apoio ao processo de Desenvolvimento de Software, a mesma não dispunha de recursos para a separação de interesses.

A partir do estudo das diferentes propostas para a modelagem orientada a aspectos, alguns pontos em comum foram identificados, indicando uma conformidade na representação de alguns conceitos da POA. Assim, foi especificada uma notação chamada UAE (*UML-based Aspect Engineering*) (GARCIA et al., 2004b) para a modelagem de sistemas orientados a aspectos, por meio da extensão da UML.

Para dar suporte a esta notação, a ferramenta de modelagem MVCASE foi estendida para apoiar o projeto orientado a aspectos por meio da notação UAE (GARCIA et al., 2004b).

[2]Disponível para download em: `http://mvcase.dev.java.net/`

4.2 Abordagem *Phoenix*

Integrando as técnicas de Reengenharia de Software e Engenharia Reversa aos mecanismos anteriormente apresentados, definiu-se uma abordagem chamada *Phoenix*, que apóia a migração de sistemas orientados a objetos para aspectos. A abordagem é composta por três passos, conforme mostra a Figura 19, utilizando a notação SADT (ROSS, 1977).

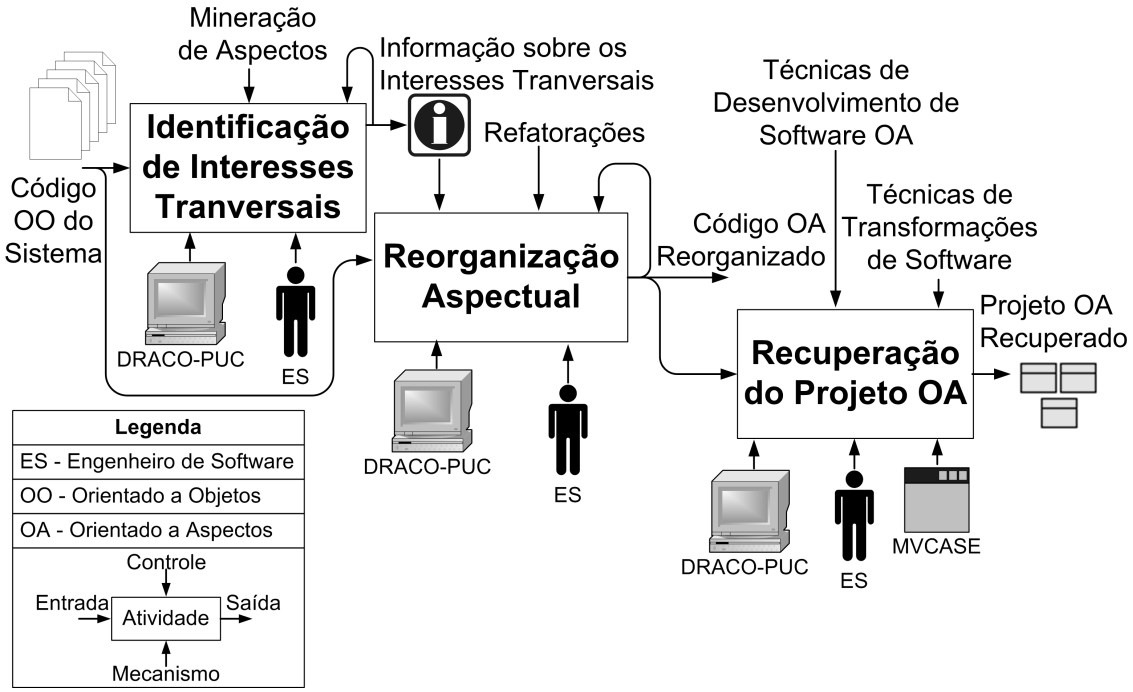


Figura 19: *Phoenix*: Uma Abordagem para Reengenharia de Software Orientada a Aspectos

No primeiro passo, parte-se do código fonte do sistema orientado a objetos para realizar a identificação dos interesses transversais. Uma vez identificados os interesses, eles podem ser extraídos, para então serem encapsulados em aspectos, por meio da aplicação de refatorações especificadas com este propósito. No passo seguinte, o Engenheiro de Software recupera o projeto do sistema orientado a aspectos em alto nível de abstração, baseado na UML, e importa esse projeto na ferramenta MVCASE. A partir de então, é possível trabalhar e realizar modificações no projeto do sistema, em um alto nível de abstração, para depois gerar o código novamente em uma linguagem orientada a aspectos. Conforme mostra a Figura 19, os principais mecanismos da abordagem são: o sistema transformacional Draco-PUC, a ferramenta MVCASE e o Engenheiro de Software. A abordagem é orientada por técnicas de Mineração de Aspectos, Refatorações, Transformação de Software e DSOA.

Segue-se uma apresentação detalhada de cada passo da abordagem, além das técnicas e mecanismos utilizados. Para a definição da abordagem foram realizados diferentes estudos de caso, de diferentes domínios (Sistema de Caixa de Banco, Sistema de Vendas pela *Web* e um sistema que

implementa o problema do Jantar dos Filósofos). Nas seções que se seguem, esses sistemas são utilizados como exemplo, para facilitar o entendimento das entradas, saídas e controles de cada passo. Os sistemas foram obtidos na Internet [3] e nenhuma documentação estava disponível além de alguns comentários no código fonte.

O entendimento dos sistemas foi obtido por meio da sua execução, onde se pode observar as suas funcionalidades. Após o entendimento, o código dos sistemas foi analisado a fim de identificar os possíveis interesses transversais, entrelaçados e dispersos através das classes do sistema.

4.2.1 Identificação de Interesses Transversais

Inicialmente, o Engenheiro de Software analisa o código orientado a objetos do sistema, procurando identificar os possíveis interesses transversais existentes. Essas informações irão servir como entrada para a mineração de aspectos, que determina onde esses interesses são situados dentro do sistema. Na abordagem *Phoenix*, a mineração de aspectos é executada usando análise de seqüências de caracteres e padrões de reconhecimento, e mineração baseada em *parser*, executada no sistema transformacional Draco-PUC[4]. A idéia também é encontrar pontos de junção estáticos, que ocorrem no contexto do programa e que se referem a interesses transversais específicos.

A mineração de aspectos baseada em *parser* é realizada por meio de transformações de software, implementadas no Draco-PUC, para identificar os interesses transversais, e utiliza uma base de conhecimento para armazenar fatos contendo informações sobre variáveis, expressões, métodos e atributos analisados no código. A Figura 20 mostra os padrões de reconhecimento que indicam a presença dos interesses de *tratamento de exceção* **(1)** e *persistência em banco de dados* **(2)**.

O interesse de tratamento de exceção **(1)** pode ser encontrado pela existência da expressão ‘‘`try-catch`’’, onde podem existir um ou mais ‘‘`catch`’’ para cada ‘‘`try`’’. Além disso pode existir uma expressão de ‘‘`finally`’’. Já o interesse de persistência em banco de dados **(2)** pode ser identificado pela utilização de determinadas classes da linguagem Java, relativas à conexão com o banco como, por exemplo a classe ‘‘`Connection`’’, e para a persistência como, por exemplo as classes ‘‘`PreparedStatement`’’, ‘‘`Statement`’’ e ‘‘`ResultSet`’’.

A Figura 21 mostra um exemplo de como a mineração baseada em *parser* pode auxiliar o Engenheiro de Software na identificação do interesse transversal de tratamento de exceção.

O *parser* reconhece a estrutura sintática ‘‘`try-catch`’’ **(1)** no código Java **(2)** e a ocorrência de

[3] `http://www.portaljava.com.br`

[4] Embora o Draco-PUC seja principalmente um sistema transformacional, ele pode atuar perfeitamente como um gerador de *parser*, conforme já ficou comprovado em trabalhos anteriores (ALVARO et al., 2003; JESUS; FUKUDA; PRADO, 1999; LEITE; SANT'ANNA; PRADO, 1996; PRADO et al., 2004).

```
statement           : 'try' '{' statement*
                      '}' catch_declaration+
                      finally_declaration?
                    ;
catch_declaration   : 'catch' '(' parameter ')' '{'
                         statement*
                      '}'
                    ;
finally_declaration: 'finally' '{' statement* '}'
                    ;
parameter           : type Identifier
                    ;
type                : name ('[]')*
                    ;
name                : Identifier++'.'
                    ;
Identifier          : [a-zA-Z_][0-9a-zA-Z_]*
                    ;
...                                              (1)
```

```
statement           : variable_declaration
                    ;
variable_declaration: var_decl ';'
                    ;
var_decl            : modifier* type
                      variable_declarator++','
                    ;
variable_declarator : Identifier ('[]')*
                      ('=' variable_initializer)?
                    ;
variable_initializer: '{' variable_initializer*
                      ','? '}'
                    ;
Identifier          : 'Connection'
                    | 'Statement'
                    | 'PreparedStatement'
                    | 'ResultSet'
...                                              (2)
```

Figura 20: Padrões de reconhecimento - Tratamento de exceção e persistência em banco de dados

```
try { [[statement* tryBody]] }
   catch ([[type exceptionType]] [[Identifier param]]) {
     [[statement* catchStatement]]
}                                                (1)
```

```
public Client addClient(...) {
Client client = null;
try {
  String sql = "SELECT ID FROM CLIENTS WHERE
USER_NAME = TRIM(?)";
  ...
}
catch (SQLException e) {
  System.out.println("Exception: " + e.toString()
+ "\n");
}
return client;
}                                                (2)
```

Figura 21: Identificando os interesses no código: Tratamento de Exceção

uma exceção do tipo `SQLException`, que indica a presença do interesse de tratamento de exceção. Esta informação é armazenada para ser consultada futuramente, a fim de ajudar o Engenheiro de Software a extrair e encapsular os interesses transversais em aspectos. Vale ressaltar que a existência da exceção do tipo `SQLException` indica também a presença do interesse transversal de persistência em banco de dados. Neste caso, cabe ao Engenheiro de Software separar o código desses dois interesses no momento em que for encapsulá-los em aspectos.

A Figura 22 mostra os trechos de código identificados (sombreados) relativos à conexão com o banco de dados no Sistema de Caixa de Banco.

Já a Figura 23 mostra os trechos de código identificados (sombreados) relativos à persistência em banco de dados, ainda no Sistema de Caixa de Banco.

Além do interesse relativo à persistência (e conexão) em banco de dados e tratamento de

```
public class BankBox {

private Connection connection;
private Vector clients;
private Vector accounts;
...
public BankBox() {
 this.init();
 this.getConnection();
 clients = new Vector();
 accounts = new Vector();
 ...
}

public Connection getConnection(){
 if (this.connection == null)
  this.setConnection();
 return this.connection;
}

private void setConnection
(Connection conn) {
 if (conn == null)
  this.setConnection();
 else
  this.connection = conn;
}
```

```
public void setConnection() {
 if (this.connection == null) {
  System.out.println("Setando
conexão...");
  String url = "jdbc:odbc:Bank";
  String username = "anonymous";
  String password = "guest";
  try {
   Class.forName(
"sun.jdbc.odbc.JdbcOdbcDriver");
   this.connection =
DriverManager.getConnection(url,
username, password);
   this.connection.setAutoCommit(true);
  } catch(ClassNotFoundException e) {
   System.err.println("Falha ao tentar
carregar o driver JDBC/ODBC:" +
e.toString() + "\n");
  } catch(SQLException e) {
   System.err.println("Não foi possível
conectar ao B.D.: " + e.toString() +
"\n");
  }
  System.out.println("Conectado com
Sucesso ao B.D. :" + url);
 }
}
```

Figura 22: Identificando os interesses no código: Conexão com o Banco de Dados

```
public Client addClient(...) {
 Client client = null;
 try {
  String sql = "SELECT ID FROM CLIENTS
WHERE USER_NAME = TRIM(?)";
  PreparedStatement stm =
this.connection.prepareStatement(sql);
  ResultSet rset = null;
  int countClients = 0;
  stm.setString(1, userName);
  rset = stm.executeQuery();
  while (rset.next()){
    countClients++;
  }
  rset.close();
  stm.close();
 } catch (SQLException e) {
  System.out.println("Exception: " +
e.toString() + "\n");
 } catch (Exception e) {
  System.out.println("Exception: " +
e.toString() + "\n");
 }
 return client;
}
```

```
public Account createAccount(...) {
 Account retValue = null;
 try {
  Statement stm = null;
  String sql = "INSERT INTO ACCOUNTS
(CLIENT_ID, ACTUAL_TYPE, ACTUAL_VALUE,
DESCR) VALUES (" + clientId + ",'" + type
+ "'," + value + ",'" + descr + "')";
  System.out.println(sql);
  stm =
this.getConnection().createStatement();
  stm.executeUpdate(sql);
  System.out.println(sql);
  stm.close();
  this.getConnection().commit();
  ...
  retValue = account;
  System.out.println("Conta criada c/
sucesso.");
 } catch (SQLException e) {
  System.out.println("ERRO: " +
e.toString());
 }
 return retValue;
}
```

Figura 23: Identificando os interesses no código: Persistência em Banco de Dados

exceção, também podem ser pesquisados outros interesses, por meio de transformações de software, como, por exemplo, os interesses de rastreamento e programação paralela. Os interesses de persistência em memória e/ou em arquivo não foram explorados na Abordagem. Na *Phoenix*, a identificação de trechos de códigos relativos ao rastreamento é realizada utilizando o seguinte conjunto de padrões de reconhecimento, conforme mostra a Figura 24.

```
statement  : 'System' '.' ('out'|'err') print_type
             '(' expression ')' ';'
           ;
print_type : 'print' | 'println'
           ;
expression : expression '.' expression
           | expression '(' arg_list? ')'
           | String | Identifier
           ;
arg_list   : expression++','
           ;
String     : \"(\\.|[^\n\"])*\"
           ;
```

Figura 24: Padrões de reconhecimento: Rastreamento

A Figura 25 mostra exemplos de trechos de códigos relativos ao interesse de rastreamento.

```
public Account createAccount(...) {
 ...
 try {
  ...
  System.out.println("Conta criada c/
sucesso.\n");
 } catch (SQLException e) { ... }
}
private void setConnection() {
 if (this.connection == null) {
  System.out.println("Setando conexão...\n");
  ...
  try {
   ...
  } catch (ClassNotFoundException e) { ... }
  catch (SQLException e) { ... }
  System.out.println("Conectado com Sucesso ao
B.D. :" + url);
 }
}
```

```
public Client loadClient(String userName) {
 Client client = new
Client(this.getConnection());
 try {
  System.out.println("bb procura: " +
userName);
  if(rSet.next()) {
   ...
  } else{
   System.out.println("Cliente não
encontrado.");
    client = null;
  }
  ...
  } catch (SQLException e) { ... }
 return client;
}
```

Figura 25: Identificando os interesses transversais: Rastreamento

O interesse transversal de programação paralela não depende de contexto para ser identificado, ele depende da classe `Thread` e da *interface* `Runnable`, presentes na biblioteca da linguagem Java. Não existindo nem a classe nem a *interface*, os padrões de reconhecimento para a identificação deste interesse não serão válidas. A Figura 26 mostra os padrões de reconhecimento para a identificação do interesse transversal de programação paralela.

A Figura 27 mostra a identificação do interesse transversal de programação paralela e tratamento de exceção, em um sistema que implementa os conceitos do problema do Jantar dos Filósofos.

As informações obtidas sobre os interesses transversais identificados, bem como sua localização no código, são armazenadas para consulta na base de conhecimento, a fim de ajudar o Engenheiro de Software a extrair e encapsular os interesses em aspectos. Os fatos armazenados na

```
class_declaration     : modifier* 'class' Identifier extends_declaration? implements_declaration?
                        '{' field_declaration* '}'
                      ;
implements_declaration: 'implements' 'Runnable'
                      ;
extends_declaration   : 'extends' 'Thread'
                      ;
field_declaration     : declaration | static_initializer
                      ;
declaration           : method_declaration | variable_declaration | type_declaration
                      ;
method_declaration    : modifier* type? Identifier '(' parameter_list? ')' ('[]')* throw_part?
                        ('{' statement* '}'  | ';')
                      ;
throw_part            : 'throws' name++','
                      ;
parameter_list        : parameter++','
                      ;
parameter             : type Identifier ('[]')*
                      ;
statement             : expression ';' | 'Thread' '.' expression
                      ;
expression            : expression '.' expression | '(' String ')' | Identifier
                      | variable_declaration | 'new' 'Thread' '(' arg_list? ')'
                      | 'new' 'Thread' ('[' expression ']')+ ('[]')* | expression '[' expression ']'
                      ;
```

Figura 26: Padrões de reconhecimento: Programação Paralela

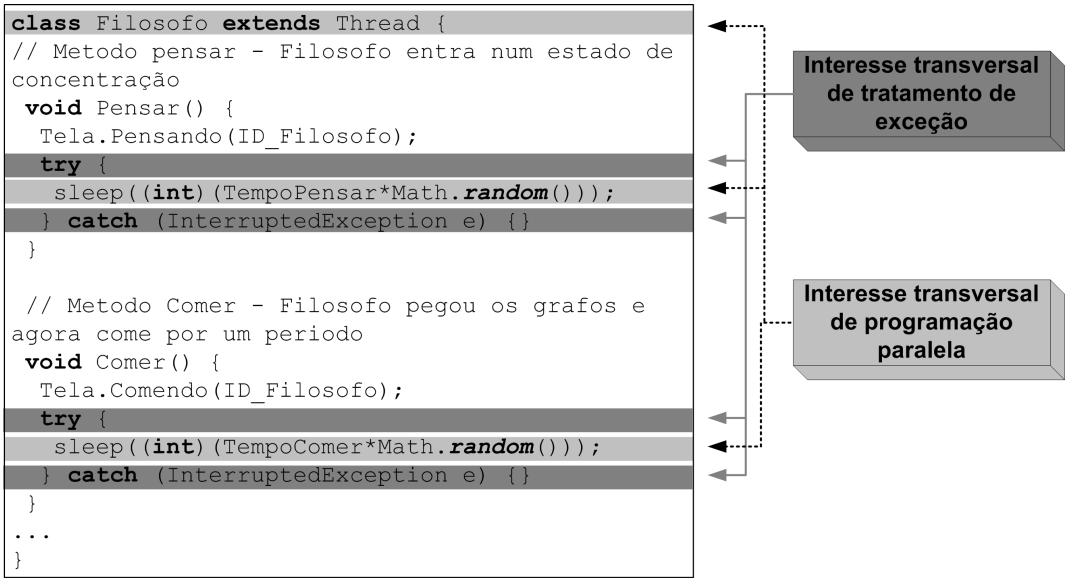


Figura 27: Identificando os interesses no código: Tratamento de Exceção e Programação Paralela

base de conhecimento são definidos como se segue:

i. *ThreadCrosscuttingConcern(<nome da classe ou interface>, <tipo do módulo(classe ou interface)>, <extends Thread ou implements Runnable>)*;

ii. *ThreadCrosscuttingConcernByVariable(<nome da classe ou interface>, <tipo do módulo (classe ou interface)>, <tipo da variável>, <nome da variável>)*;

iii. *ExceptionHandlingCCByThrow(<nome da classe>, <identificador do método>, <nome do método>, <tipo da exceção>)*;

iv. *ExceptionHandlingCCByTryCatch(<nome da classe ou interface>, <tipo do módulo (classe ou interface)>, <identificador do método>, <nome do método>, <tipo de parâmetro do catch>, <nome do parâmetro do catch>)*;

v. *DataBasePersistenceCCByVariable(<nome da classe ou interface>, <tipo do módulo (classe ou interface)>, <tipo da variável>, <nome da variável>)*;

vi. *DataBasePersistenceCCByMethod(<nome da classe ou interface>, <tipo do módulo (classe ou interface)>, <tipo de retorno do método>, <nome do método>)*;

vii. *DataBasePersistenceCCByException(<nome da classe ou interface>, <tipo do módulo (classe ou interface)>, <identificador do método>, <nome do método>, <tipo de parâmetro do catch>, <nome do parâmetro do catch>)*;

viii. *TraceCrosscuttingConcern(<nome da classe>, <identificador do método>, <nome do método>, <tipo da mensagem>, <mensagem>)*;

Os demais fatos, relacionados às informações das classes, interfaces, métodos e atributos, armazenados na base de conhecimento são definidos como se segue:

i. *Package(<nome do pacote>)*;

ii. *Class(<modificador>, <nome da classe>, <classe estendida>, <implementa uma ou mais interface>)*;

iii. *ClassModifier(<nome da classe>, <identificador do modificador>, <modificador>)*;

iv. *Implements(<nome da classe>, <identificador do implements>, <nome da interface implementada>)*;

v. *Interface(<modificador>, <nome da interface>, <estende um ou mais classes>)*;

vi. *InterfaceModifier(<nome da interface>, <identificador do modificador>, <modificador>)*;

vii. *Extends(<nome da interface>, <identificador do extend>, <nome da classe estendida>)*;

viii. *Method(<nome da classe>, <identificador do método>, <modificador>, <tipo de retorno>, <nome do método>, <possui parâmetros>, <lança exceção>)*;

ix. *MethodModifier(<nome da classe>, <identificador do método>, <identificador do modificador>, <modificador>)*;

x. *MethodParameter(<nome da classe>, <identificador do método>, <nome do método>, <tipo do parâmetro>, <nome do parâmetro>, <possui colchetes>)*;

xi. *Variable(<nome da classe>, <modificador>, <tipo>, <nome>, <valor inicial>)*;

xii. *VariableModifier(<nome da classe>, <nome da variável>, <identificador do modificador>, <modificador>)*;

A Figura 28 mostra alguns fatos armazenados na base de conhecimento referentes à classe *Principal*.

```
Class(modifier,Principal,JPanel,none).
ClassModifier(Principal,1,public).
Variable(Principal,modifier,File,currentFile,none).
VariableModifier(Principal,currentFile,1,private).
Variable(Principal,modifier,Simulation,simulation,none).
VariableModifier(Principal,simulation,1,private).
Method(Principal,7,modifier,void,jbInit,none,throw).
MethodModifier(Principal,7,1,private).
ExceptionHandlingCCByThrow(Principal,class,7,jbInit,1,Exception).
Method(Principal,9,modifier,void,openFile,none,none).
MethodModifier(Principal,9,1,public).
Variable(Principal,modifier,int,result,initializer).
Variable(Principal,modifier,File,tmp,initializer).
Variable(Principal,modifier,BufferedReader,bin,initializer).
Variable(Principal,modifier,String,input,none).
ExceptionHandlingCCByTryCatch(Principal,class,9,openFile,Exception,ex).
```

Figura 28: Fatos armazenados na base de conhecimento

Entretanto deve-se reforçar que a mineração baseada em *parser*, assim como em seqüências de caracteres e na análise de padrões de reconhecimento, é uma forma de auxiliar a identificação dos interesses, que não impede que os interesses também sejam identificados manualmente, por inspeção do código.

Na Figura 29 os trechos de códigos relacionados com a atualização da tela com as informações do cliente estão sombreados. Para identificar os interesses por meio da inspeção de código, é necessário realizar um estudo sobre a execução e o comportamento do sistema.

Após a identificação, parte-se para o passo subseqüente, a Reorganização Aspectual. Antes porém, é importante ressaltar que alguns trechos de código podem ter sido identificados como pertencendo a dois ou mais interesses transversais como, por exemplo, tratamento de exceção e rastreamento, onde em muitos casos tem-se a impressão de mensagens por meio do ‘‘`System.out.println`’’ dentro das expressões de ‘‘`catch`’’. Neste caso, cabe ao Engenheiro de Software analisar, manualmente, cada trecho em conflito e definir qual o interesse que ele melhor

```
private Vector loadClients() {
 try {
  String sql = "SELECT * FROM CLIENTS ORDER BY ID,FULL_NAME";
  Statement stm = this.connection.createStatement();
  ResultSet rSet = null;
  rSet = stm.executeQuery(sql);
  this.clients.clear();
  while (rSet.next()) {
   Client client = new Client(this.getConnection());
   client.setId(rSet.getInt("ID"));
   client.setUserName(rSet.getString("USER_NAME"));
   client.setFullName(rSet.getString("FULL_NAME"));
   ...
   this.clients.add(client);
  }
  System.out.println("Cliente(s) carregado(s): ["+clients.size()+"]\n");
  rSet.close();
  stm.close();
 } catch (SQLException e) {
  System.err.println("\n ERRO SQL: "+e.toString()+ "\n");
 }
clientsTable.display(clients);
return this.clients;
}
```

Figura 29: Identificando interesses no código: Atualização da tela com as informações do Cliente

representa.

No segundo passo da abordagem *Phoenix*, o código fonte é organizado de acordo com os princípios da POA, separando os interesses transversais (requisitos funcionais e/ou não-funcionais) em aspectos, e as regras de negócio (requisitos funcionais) em classes com seus respectivos métodos e atributos. Essa separação é auxiliada pelo catálogo de refatorações orientadas a aspectos (GARCIA et al., 2004).

4.2.2 Reorganização Aspectual

Após a identificação dos interesses transversais, o Engenheiro de Software utiliza refatorações orientadas a aspectos para extrair e encapsular esses interesses em aspectos. Essas refatorações consideram a natureza entrelaçada do código do sistema. Assim, a transferência de membros individuais das classes para um aspecto não deve ser feita isoladamente. Na maioria dos casos, são parte de um conjunto de transferências que compreendem todos os elementos da implementação do interesse que está sendo extraído. Tais interesses incluem, tipicamente, os múltiplos fragmentos do código dispersos nas unidades modulares (por exemplo métodos, classes, pacotes). Conseqüentemente, mais de uma refatoração deve ser aplicada para extrair um interesse particular.

Considere o seguinte código fonte do Sistema de vendas pela Web:

```
import java.lang.reflect.*;
public class ShoppingCart {
  private List items = new Vector();
```

```
  private java.io.PrintWriter logWriter = System.err;
  public void addItem(Item item) {
      logWriter.println("Log:"+this.getClass().getName());
      items.add(item);
  }
  public void removeItem(Item item) {
      logWriter.println("Log:"+this.getClass().getName());
      items.remove(item);
  }
  public void empty() {
      logWriter.println("Log:"+this.getClass().getName());
      items.clear();
  }
}
```

Este código mistura a lógica do negócio com o requisito de *logging*. O código de *logging* (sombreado) deve ser extraído para um aspecto. Para fazer isto, as refatorações podem ser utilizadas, auxiliando a mover atributos, partes de código e métodos inteiros que são relacionados ao *logging* no novo aspecto.

Para facilitar a compreensão na especificação das refatorações foi escolhido um formato semelhante ao utilizado por Fowler em (FOWLER et al., 1999). Será utilizado também, assim como Fowler, um único código no exemplo de aplicação das refatorações. A especificação compreende os seguintes elementos: o nome da refatoração; uma descrição da situação na qual ela é necessária e o que ela propõe; uma descrição da ação recomendada; pré-condições para sua aplicação, quando existir; o mecanismo de aplicação da refatoração e um código de exemplo.

A primeira refatoração aplicada é a **EXTRACT FIELD TO ASPECT**. Ela deve ser aplicada quando uma classe tem um atributo relacionado a um interesse transversal que está implementado ou está sendo extraído para um aspecto. O código a seguir mostra o atributo relacionado ao interesse de *logging* em sombreado.

```
import java.lang.reflect.*;
public class ShoppingCart {
  private List items = new Vector();
  private java.io.PrintWriter logWriter = System.err;
  public void addItem(Item item) {
      logWriter.println("Log:"+this.getClass().getName());
      items.add(item);
  }
  public void removeItem(Item item) {
      logWriter.println("Log:"+this.getClass().getName());
      items.remove(item);
  }
  public void empty() {
      logWriter.println("Log:"+this.getClass().getName());
```

```
    items.clear();
  }
}
```

A solução é mover o atributo como uma declaração intertipo para o aspecto que implementa o interesse.

Motivação: Extraindo o atributo para um aspecto, o problema de entrelaçamento de código é reduzido, já que o aspecto irá conter todos os atributos relacionados a um único interesse, e nenhum atributo relacionado a esse interesse estará situado fora do aspecto.

Mecanismo.

- Se o atributo for definido como público, considere o uso do *Encapsulate Field* ((FOWLER et al., 1999), p.206) antes que esta refatoração seja aplicada.
- Mova a declaração do atributo para o aspecto, incluindo a declaração de valor inicial, se houver uma. Isto deve ser feito de acordo com a sintaxe da declaração intertipos da linguagem orientada a aspectos utilizada.
- Analise se deve ser adicionada a declaração de pacote.
- Modifique a visibilidade do atributo para *public*. A visibilidade pode ser modificada depois que a refatoração for aplicada. Se ainda houver a necessidade de manter o código relacionado ao atributo na classe, considere usar o *Self Encapsulate Field* ((FOWLER et al., 1999), p.146).
- Verifique todos os conjuntos de pontos de junção que contém declarações *"within()"*, pois devem ser atualizadas após a aplicação desta refatoração.
- Compile e teste.

A **EXTRACT FIELD TO ASPECT** pode ser utilizada para extrair atributos relacionados ao interesse de *logging* para o aspecto. Após a refatoração ser aplicada, tem-se:

```
public aspect LoggingAspect {
  private java.io.PrintWriter ShoppingCart.logWriter = System.err;
  pointcut loggedMethods(ShoppingCart shoppingcart): this(shoppingcart)
  && (execution(void ShoppingCart.*(..)));
  before(ShoppingCart shoppingcart): loggedMethods(shoppingcart) {
    logWriter.println("Log:"+shoppingcart.getClass().getName());
  }
}
```

... após a **EXTRACT FIELD TO ASPECT** ser aplicada, alguns métodos que se relacionam àqueles atributos também devem ser movidos para o aspecto, desde que provavelmente relacionem-se ao mesmo interesse que está sendo extraído. Para extrair estes métodos, assim como outros métodos que relacionam-se a este interesse, pode-se utilizar **EXTRACT METHOD TO ASPECT**.

A **EXTRACT METHOD TO ASPECT** é aplicada quando existem métodos na classe que se relacionam a um interesse transversal implementado ou que esteja sendo extraído para um aspecto. O problema é como extrair o método para o aspecto, já que o método tem sua própria visibilidade, valor de retorno e faz parte do contexto da classe.

```
import java.lang.reflect.*;
public class ShoppingCart {
  private List items = new Vector();
  private void addLogMessage(String message) {
     System.out.println("Log "+(logNumber++)+":"+message);
  }
  public void addItem(Item item) {
     addLogMessage(this.getClass().getName());
     items.add(item);
  }
  public void removeItem(Item item) {
     addLogMessage(this.getClass().getName());
     items.remove(item);
  }
  public void empty() {
     addLogMessage(this.getClass().getName());
     items.clear();
  }
}
```

A solução seria mover o método para o aspecto que implementa o interesse transversal, como uma declaração intertipos.

Motivação: Um método que relaciona-se com um interesse particular deve estar situado dentro do aspecto que implementa este interesse, para reduzir o entrelaçamento e espalhamento de código.

Mecanismo.

- Mover a declaração do método que está na classe para o aspecto. Isto deve ser feito de acordo com a sintaxe da declaração intertipos da linguagem orientada a aspectos utilizada.
- Analisar se deve ser adicionada a declaração de pacote.
- Verificar todos os *conjuntos de pontos de junção* com a declaração *"within()"*, que devem ser atualizados após a aplicação desta refatoração.

- Compilar e testar.

A **EXTRACT METHOD TO ASPECT** pode ser utilizada para extrair o método relacionado ao interesse de *logging* para o aspecto. Após a sua aplicação tem-se:

```
public aspect LoggingAspect {
  private void ShoppingCart.addLogMessage(String message) {
     System.out.println("Log :"+message);
  }
  pointcut loggedMethods(ShoppingCart shoppingcart): this(shoppingcart)
  && (execution(void ShoppingCart.*(..)));
  before(ShoppingCart shoppingcart): loggedMethods(shoppingcart) {
    addLogMessage(shoppingcart.getClass().getName());
  }
}
```

. . . após a **EXTRACT METHOD TO ASPECT** ser aplicada, não existe mais nenhum método que esteja relacionado ao interesse que está sendo extraído. Entretanto, as chamadas a esses métodos permanecem localizadas em outros métodos que não foram extraídos para o aspecto. Pode também existir trechos de código que estão relacionados a esse interesse, mas não estão agrupados como métodos. Para extrair essas chamadas de método e outros trechos de código, utiliza-se a **EXTRACT CODE TO ADVICE**.

A **EXTRACT CODE TO ADVICE** é aplicada quando algum trecho de código está relacionado a um interesse que pode ser implementado como um aspecto. Esse código não está separado em um único método, mas sim misturado com outros códigos relacionados a outros interesses.

A solução seria extrair o trecho de código para um adendo

Motivação: As chamadas ao método que ocorrem em diversas classes são geralmente sujeitas a esta refatoração. A separação correta dos interesses pode melhorar a manutenibilidade e o reuso do software.

Entretanto, antes de realizar a extração de parte de código do método para um adendo no aspecto, é necessário realizar uma cuidadosa análise do corpo do método, para especificar os conjuntos de pontos de junção que irão capturar os apropriados pontos de junção. Se o código não oferecer nenhum ponto de junção apropriado, outras refatorações poderão ser aplicadas para corrigir este problema.

Mecanismo.

- Criar um novo adendo.

- Mover o trecho de código que está sendo extraído para o corpo do adendo.
- Adicionar o código para implementar o contexto do adendo.
- Substituir as referências ao *"this"* (o objeto atual) pela variável capturada no contexto do ponto de junção.
- Analisar o trecho de código extraído, procurando por referências a variáveis locais no escopo do método, incluindo parâmetros e atributos locais. Quaisquer declarações de variáveis temporárias, utilizadas somente no trecho de código extraído, podem ser substituídas no corpo do adendo.
- Compilar o código e testar.

Exemplo.

Esta refatoração pode ser utilizada para extrair o trecho de código relacionado à execução do *addItem* para o aspecto. Após a aplicação da refatoração tem-se:

```
import java.lang.reflect.*;
public class ShoppingCart {
  private List items = new Vector();
  public void addItem(Item item) {
     items.add(item);
  }
  public void removeItem(Item item) {
     System.out.println("Log:"+this.getClass().getName());
     items.remove(item);
  }
  public void empty() {
     System.out.println("Log:"+this.getClass().getName());
     items.clear();
  }
}
```

O código relacionado com o *logging* foi extraído para o aspecto `Foo`:

```
public aspect Foo {
  before(ShoppingCart shoppingcart): this(shoppingcart) &&
  (execution(void ShoppingCart.addItem(..))) {
    System.out.println("Log:"+shoppingcart.getClass().getName());
  }
}
```

Nota-se que o contexto referente à variável *"this"* também foi movido para o aspecto.

... após essa refatoração ter sido aplicada, ainda existem chamadas aos métodos que relacionam-se ao interesse de *logging*. Assim, aplica-se a **EXTRACT CODE TO ADVICE** nos outros métodos afetados na classe. Então, a classe fica desta forma:

```
import java.lang.reflect.*;
public class ShoppingCart {
  private List items = new Vector();
  public void addItem(Item item) {
     items.add(item);
  }
  public void removeItem(Item item) {
     items.remove(item);
  }
  public void empty() {
     items.clear();
  }
}
```

E o aspecto:

```
public aspect Foo {
  before(ShoppingCart shoppingcart): this(shoppingcart) &&
  (execution(void ShoppingCart.addItem(..)))
  || (execution(void ShoppingCart.removeItem(..)))
  || (execution(void ShoppingCart.empty(..))) {
    System.out.println("Log:"+shoppingcart.getClass().getName());
  }
}
```

... após a aplicação da **EXTRACT CODE TO ADVICE** a diversos métodos, a declaração dos pontos de junção cresce e torna-se altamente acoplada com o corpo do adendo. A fim de melhorar a legibilidade e diminuir esse acoplamento, **EXTRACT POINTCUT DEFINITION** pode ser utilizada.

A **EXTRACT POINTCUT DEFINITION** deve ser aplicada quando a definição dos pontos de junção estão altamente acopladas ao código do adendo.

A solução seria separar essa definição em conjuntos de pontos de junção.

Motivação: Pode ser definido um conjunto de pontos de junção separado, definindo os pontos de junção e o contexto correto. A separação entre as definições do conjunto de pontos de junção e os adendos torna o código mais legível e flexível, já que as modificações podem ser executadas em um único lugar. Além disso, melhora-se reuso do código, visto que uma única definição do conjunto de pontos de junção pode ser utilizada por diversos adendos.

Mecanismo.

- Crie um conjunto de pontos de junção que capture os pontos de junção apropriados.
- Assegure-se de que o conjunto de pontos de junção capture todo o contexto requerido pelo trecho de código. Verifique se o código extraído se refere a declarações *"this"* ou *"super"*. Os tipos de predicados mais utilizados definidos nos conjuntos de pontos de junção usam referências ao objeto que recebe uma mensagem e chamadas a um método específico, ou uma referência ao objeto atual combinado com a execução do método, atributo lido e operações de escrita, entre outros.
- Compile e teste.

Exemplo.

Após extrair a definição do conjunto de pontos de junção:

```
public aspect Foo {
  pointcut test(ShoppingCart shoppingcart): this(shoppingcart) &&
  (execution(void ShoppingCart.addItem(..)))
  || (execution(void ShoppingCart.removeItem(..)))
  || (execution(void ShoppingCart.empty(..)))
  );
  before(ShoppingCart shoppingcart): test(shoppingcart) {
    System.out.println("Log:"+shoppingcart.getClass().getName());
  }
}
```

...após a aplicação da **EXTRACT POINTCUT DEFINITION**, a declaração dos pontos de junção podem tornar-se muito complexas, com informações em excesso. Um predicado equivalente pode ser utilizado em vez dos pontos de junção individuais. Para isto, utiliza-se a **COLLAPSE POINTCUT DEFINITION**.

A **COLLAPSE POINTCUT DEFINITION** deve ser aplicada quando a declaração do conjunto de pontos de junção está muito complexa, com excesso de informação.

Um predicado equivalente pode ser utilizado ao invés de pontos de junção individuais.

Motivação: Quando o número de pontos de junção aumenta, a manutenibilidade dos conjuntos de pontos de junção torna-se mais difícil. A definição aumenta enquanto as novas classes e os métodos são criados e as referências são adicionadas ao aspecto. Expressões regulares podem ser utilizadas para expressar os conjuntos de pontos de junção.

Mecanismo.

- Verificar quantos pontos de junção determináveis são afetados atualmente pelo conjunto de pontos de junção.
- Converter a declaração do ponto de junção em uma expressão regular, pelo uso dos coringas a atributos lógicos disponibilizados pela linguagem orientada a aspectos.
- Compilar e testar.
- Verificar se o número de pontos afetados é o mesmo que foi determinado anteriormente.

Exemplo.

Em seguida à aplicação da COLLAPSE POINTCUT DEFINITION:

```
public aspect Foo {
  pointcut test(ShoppingCart shoppingcart): this(shoppingcart) &&
  (execution(void ShoppingCart.*(..)));
  before(ShoppingCart shoppingcart): test(shoppingcart) {
    System.out.println("Log:"+shoppingcart.getClass().getName());
  }
}
```

. . . após formatar a definição, pode-se identificar que o nome do conjunto de pontos de junção não ajuda a identificar qual a sua real função. Assim, para fornecer um nome significativo, pode-se aplicar a **RENAME POINTCUT**.

A **RENAME POINTCUT** deve ser aplicada quando o nome do conjunto de pontos de junção não revela o seu real objetivo.

A solução seria modificar o nome do conjunto de pontos de junção.

Motivação: Os nomes são um dos melhores mecanismos para expressar o objetivo de uma determinada funcionalidade. Se existirem nomes ímpares às classes, aos métodos e aos aspectos, seu código torna-se mais difícil para compreender e manter. O ideal é fornecer nomes significativos para melhorar a qualidade do código.

Mecanismo.

- Crie um novo conjunto de pontos de junção com um nome próprio.
- Copie a declaração do antigo conjunto de pontos de junção para o novo.
- Modifique o antigo conjunto de pontos de junção para apontar para o novo (se possuir poucas referências, pode-se pular este passo)

- Compile e teste.
- Modifique as referências do antigo conjunto de pontos de junção nos adendos e outros conjuntos de pontos de junção para o novo.
- Compile e teste para cada modificação.
- Remova a declaração do antigo conjunto de pontos de junção.
- Compile e teste.

Exemplo. Após renomear o conjunto de pontos de junção tem-se:

```
public aspect Foo {
  pointcut loggedMethods(ShoppingCart shoppingcart):
  this(shoppingcart) && (execution(void ShoppingCart.*(..)));
  before(ShoppingCart shoppingcart): loggedMethods(shoppingcart) {
    System.out.println("Log:"+shoppingcart.getClass().getName());
  }
}
```

Após ter extraído o interesse relacionado ao *logging* para o aspecto `Foo`, foi identificado que esse aspecto deveria ser rebatizado com um nome mais significativo, por meio da **RENAME ASPECT**.

A **RENAME ASPECT** deve ser aplicada quando o nome do aspecto não revela sua real funcionalidade.

A solução seria modificar o nome do aspecto.

Motivação: Uma boa prática é nomear classes, aspectos, métodos, etc, a fim de melhorar a legibilidade do código. Se para descobrir a função de um método ou classe for necessário analisar todo o código fonte, esta é uma situação onde tem-se um candidato a aplicação desta refatoração.

Mecanismo.

- Verificar se o nome do aspecto está sendo utilizado por um super-aspecto ou por um sub-aspecto. Em caso positivo, renomear também as referências no sub e super-aspectos.
- Modificar o nome do aspecto.
- Compilar e testar.

Após renomear o aspecto:

```
public aspect LoggingAspect {
  pointcut loggedMethods(ShoppingCart shoppingcart): this(shoppingcart)
  && (execution(void ShoppingCart.*(..)));
  before(ShoppingCart shoppingcart): loggedMethods(shoppingcart) {
    System.out.println("Log:"+shoppingcart.getClass().getName());
  }
}
```

... após o aspecto ser extraído, com seus atributos, métodos e adendos, e propriamente batizado, algumas características podem ser separadas, como, por exemplo, as definições dos conjuntos de pontos de junção da implementação, utilizando conjuntos de pontos de junção e aspectos abstratos, para aumentar a reusabilidade, flexibilidade e legibilidade.

Para prover e estimular o reuso pode-se usar a **PULL UP ADVICE/POINTCUT**. Ela deve ser aplicada quando um aspecto tem as funcionalidades que poderiam estar sendo utilizadas por diversos aspectos relacionados.

A solução seria mover essas funcionalidades para um super-aspecto.

Motivação: A duplicação de código é um dos responsáveis pelos elevados custos na manutenção. O uso de herança pode ajudar a evitar esse problema.

Mecanismo.

- Analisar os conjuntos de pontos de junção/adendos para assegurar-se de que são idênticos.
- Criar um novo conjunto de pontos de junção abstrato no super-aspecto com a mesma assinatura que nos sub-aspectos.
- Criar um novo adendo no super-aspecto e copiar o corpo do antigo adendo para o definido no super-aspecto.
- Remover o antigo adendo.
- Compilar e testar.

Exemplo.

Após aplicar a **PULL UP ADVICE/POINTCUT**, tem-se:

```
abstract aspect AbstractLoggingAspect {
  abstract pointcut loggedMethods(ShoppingCart shoppingcart);
  before(ShoppingCart shoppingcart): loggedMethods(shoppingcart) {
    System.out.println("Log:"+shoppingcart.getClass().getName());
```

```
  }
}

public aspect LoggingAspect extends AbstractLoggingAspect {
  pointcut loggedMethods(ShoppingCart shoppingcart): this(shoppingcart)
  && (execution(void ShoppingCart.*(..)));
}
```

...algumas vezes, separar funcionalidades por toda a hierarquia dos aspectos não é uma alternativa eficiente. Neste caso, o processo exatamente oposto pode ser aplicado usando a **COLLAPSE ASPECT HIERARCHY**.

A **COLLAPSE ASPECT HIERARCHY** deve ser aplicada quando tem-se um aspecto e um super-aspecto que são completamente similares, ou não vale a pena o custo de ter um super-aspecto.

A solução é combinar o aspecto e o seu super-aspecto em um só.

Motivação: Separar funcionalidades em aspectos e super-aspectos é muito útil quando se quer manter essas funcionalidades disponíveis a outros aspectos. Entretanto, esta separação tem um custo, ou na legibilidade ou na manutenibilidade, já que existem duas unidades modulares a serem consideradas: o super-aspecto e o sub-aspecto. Se esses custos não forem compensados por ganhos no reuso, é melhor que essas funcionalidades sejam agrupadas em uma única unidade modular.

Mecanismo.

- Escolher qual aspecto será removido: o sub-aspecto ou o seu super-aspecto.
- Usar a PULL UP ADVICE/POINTCUT, PULL UP FIELD((FOWLER et al., 1999), p.320) e PULL UP METHOD((FOWLER et al., 1999), p.322), PUSH DOWN METHOD((FOWLER et al., 1999), p.328) e PUSH DOWN FIELD((FOWLER et al., 1999), p.329) caso necessário.
- Compilar e testar a cada operação de PUSH/PULL.
- Modificar as referências às classes que serão removidas para utilizar o aspecto combinado, resultante da aplicação da refatoração.
- Remover o aspecto selecionado para exclusão.
- Compilar e testar.

Após aplicar a **COLLAPSE ASPECT HIERARCHY**, tem-se:

```
public aspect LoggingAspect {
```

```
  pointcut loggedMethods(ShoppingCart shoppingcart):this(shoppingcart)
  && (execution(void ShoppingCart.*(..)));
  before(ShoppingCart shoppingcart): loggedMethods(shoppingcart) {
    System.out.println("Log:"+shoppingcart.getClass().getName());
  }
}
```

. . . após desfazer a hierarquia dos aspectos, pode-se inferir que o adendo está tão legível quanto o nome dos conjuntos de pontos de junção relacionados. Assim, para fornecer um nome significativo, pode-se aplicar a **INLINE POINTCUT DEFINITION**.

A **INLINE POINTCUT DEFINITION** deve ser aplicada quando tem-se uma situação em que um conjunto de pontos de junção é utilizado com um adendo somente e sua funcionalidade é completamente óbvia. Não há nenhuma razão separar a definição do conjunto de pontos de junção do adendo.

A solução é mover o predicado que contém os pontos de junção afetados para a assinatura do adendo.

Motivação: Separar a definição de um conjunto de pontos de junção dos adendos é interessante quando resulta em aumento do reuso e maior legibilidade e flexibilidade (vide refatoração **EXTRACT POINTCUT DEFINITION**). Porém, nos casos onde este conjunto é muito simples, e é utilizado somente por um único resultado de um adendo, esta definição extra é desnecessária.

Mecanismo.

- Verificar se o conjunto de pontos de junção/adendo não é estendido em sub-aspectos e se não está definido como abstrato.
- Buscar todas as referências ao conjunto de pontos de junção.
- Substituir as referências à definição do conjunto de pontos de junção.
- Remover o predicado do conjunto de pontos de junção e definí-lo como abstrato.
- Compilar e testar.
- Remover o conjunto de pontos de junção.
- Compilar e testar.

Exemplo.

Após a aplicação da **INLINE POINTCUT DEFINITION**, tem-se:

```
public aspect Foo {
  before(ShoppingCart shoppingcart): this(shoppingcart) &&
   (execution(void ShoppingCart.*(..))) {
    System.out.println("LOG:"+shoppingcart.getClass().getName());
  }
}
```

As refatorações apresentadas são aplicadas manualmente, porém, sabe-se que existe uma grande probabilidade de erros serem inseridos pelo Engenheiro de Software durante esta atividade. Para automatizar e assegurar a não inserção de erros durante a aplicação das refatorações, podem ser implementadas transformações de software específicas para tal.

A Figura 30 mostra um exemplo de transformação para extrair o interesse de tratamento de exceção do código fonte, em Java. A estrutura sintática da expressão ‘‘try-catch’’ **(1)** é identificada por um padrão de leitura, chamado *LHS*, no transformador ‘‘RefactoringExceptionByTryCatch’’ **(2)**. Após a identificação da declaração ‘‘try-catch’’ o ponto de controle POST-MATCH é executado. Em seguida, o padrão de escrita, chamado *RHS*, cria um aspecto responsável pelo tratamento de exceção **(3)**. O Engenheiro de Software é responsável por especificar o nome do aspecto e do conjunto de pontos de junção, ou implementar um mecanismo nas transformações de software para atribuir esses nomes.

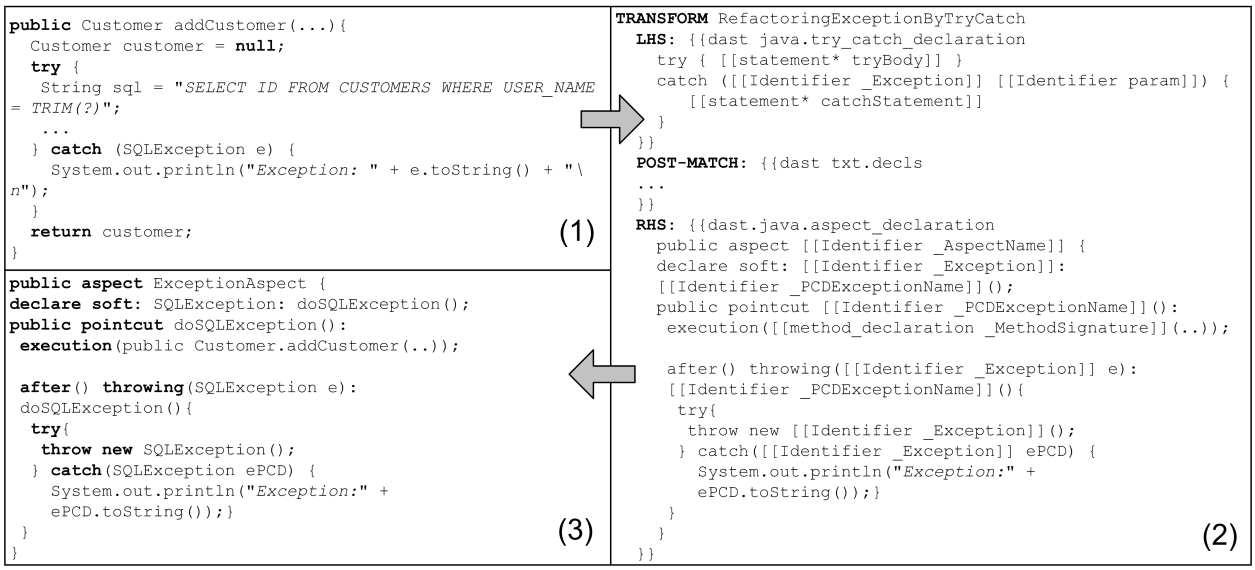


Figura 30: Refatoração orientada a aspectos para extrair o Tratamento de Exceção

Nós acreditamos que assim como algumas refatorações orientadas a objetos propostas por Fowler (1999) são passíveis de implementação (automatização), algumas refatorações orientadas a aspectos também o são como, por exemplo, **RENAME ASPECT**, **EXTRACT FIELD TO ASPECT** e **EXTRACT METHOD TO ASPECT**. A implementação para a extração dos trechos de código relativos ao interesse de tratamento de exceção é o primeiro passo para alcançar esse

objetivo.

Após obter o código do sistema organizado segundo a Orientação a Aspectos, com os interesses transversais encapsulados em aspectos, e a sua validação, parte-se para o último passo da abordagem para recuperar o projeto orientado a aspectos do sistema.

4.2.3 Recuperação do Projeto Orientado a Aspectos

Neste passo, o Engenheiro de Software, utilizando transformações de software no sistema transformacional Draco-PUC, obtém o projeto orientado a aspectos, em descrições UML. As transformações mapeiam descrições em uma linguagem de programação, correspondendo ao código orientado a aspectos reorganizado, para descrições em uma linguagem de modelagem, que podem ser importadas em uma ferramenta de modelagem com suporte à modelagem segundo o paradigma OA, como a MVCASE, por exemplo. Com o projeto recuperado, o Engenheiro de Software pode editar os modelos recuperados na ferramenta, adicionando pequenas correções e refinamentos. A MVCASE utiliza a extensão da UML, denominada UAE (*UML-based Aspect Engineering*), que possibilita a representação dos conceitos do DSOA (GARCIA et al., 2004b). Maiores informações sobre recuperação de projeto usando transformações podem ser obtidas em (ALVARO et al., 2003; GARCIA et al., 2004c, 2005; PRADO et al., 2004).

A Figura 31 mostra um exemplo de recuperação de projeto orientado a aspectos usando transformações de software, implementadas no Draco-PUC. O código orientado a aspectos **(1)** é analisado por transformações **(2)**, que são responsáveis por mapear o código em descrições em uma linguagem de modelagem (no caso, foi utilizado XMI (OMG, 2002)) **(3)**. Essas descrições são importadas na MVCASE, ficando disponíveis para edição **(4)**.

No padrão de leitura *LHS*, em **(2)**, o transformador reconhece a declaração do aspecto, no domínio da linguagem *AspectJ*. Após a identificação da declaração do aspecto, o ponto de controle *POST-MATCH* é executado. Em seguida, o padrão de escrita *RHS* é executado por meio de uma *TEMPLATE*, que persiste o aspecto em especificações da linguagem de modelagem XMI. Posteriormente, os membros do aspecto também são persistidos em especificações XMI.

Em seguida, o Engenheiro de Software, utilizando a ferramenta MVCASE, importa as descrições XMI contendo as definições dos aspectos e das classes para visualizar o projeto do sistema orientado a aspectos. Esse projeto é representado usando o diagrama de classes com a notação específica para DSOA.

Para importar o projeto recuperado em descrições XMI na MVCASE é necessário que o *plug-in AOSD* esteja adicionado à ferramenta. Este *plug-in*, que foi construído neste trabalho para

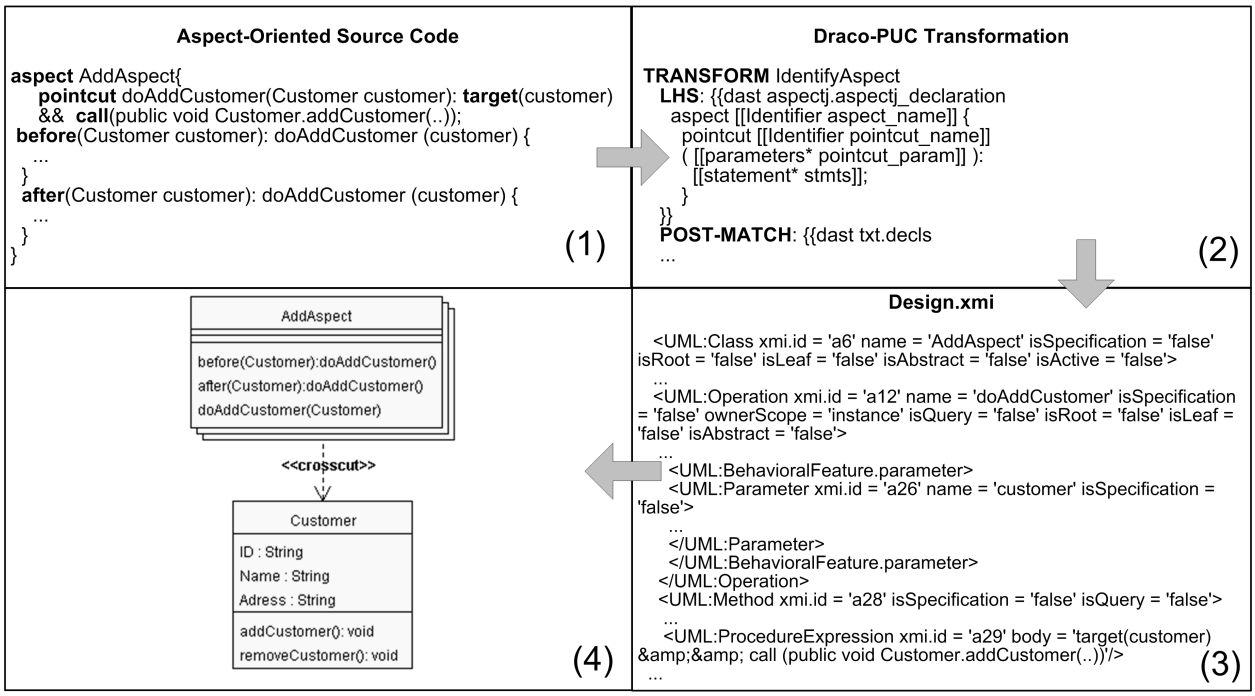


Figura 31: Recuperação do Projeto Orientado a Aspectos

incorporar à ferramenta as extensões da POA. Para criar um novo projeto orientado a aspectos, o Engenheiro de Software deve, a partir do projeto já criado na MVCASE, inserir um novo diagrama na visão lógica (*Logical View*), por meio da seqüência de *ações*: *Botão direito do mouse -> AOSD -> Create Aspect Diagram*. A seguir, um novo diagrama é criado, permitindo ao Engenheiro de Software utilizar os estereótipos implementados para representar as construções da POA, são eles: <<*aspect*>>, <<*crosscut*>> e <<*inter-type*>>, conforme mostra a Figura 32.

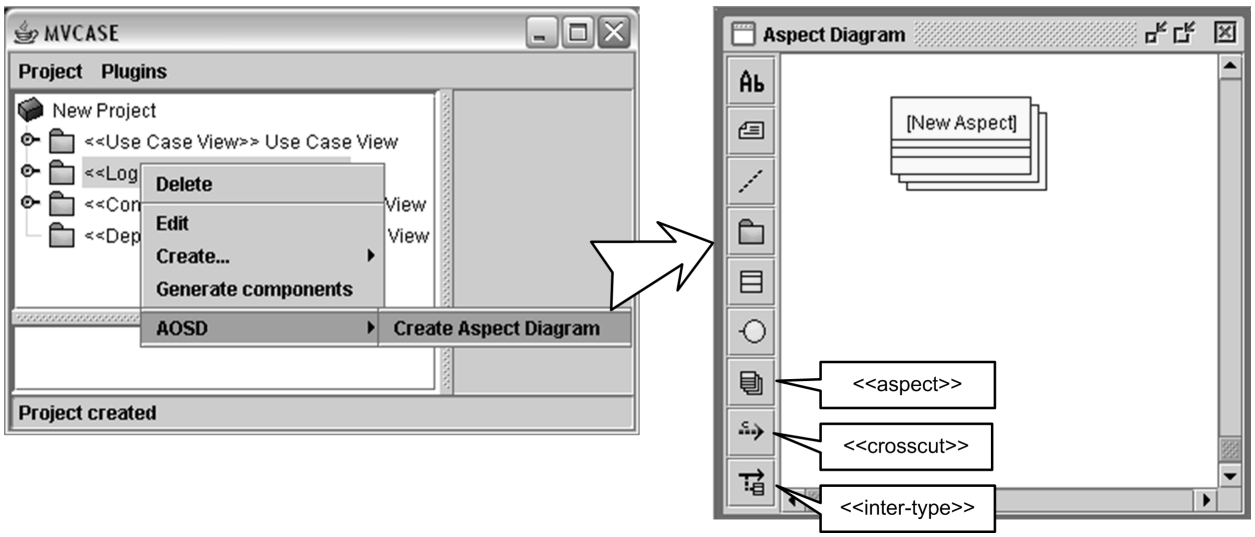


Figura 32: Extensão da MVCASE para o Projeto Orientado a Aspectos

No caso da Abordagem *Phoenix*, as transformações de software para recuperar o projeto orientado a aspectos do sistema geram as especificações em XMI já com o projeto e o diagrama de

aspectos. A Figura 33 mostra a janela para especificação de um aspecto na MVCASE.

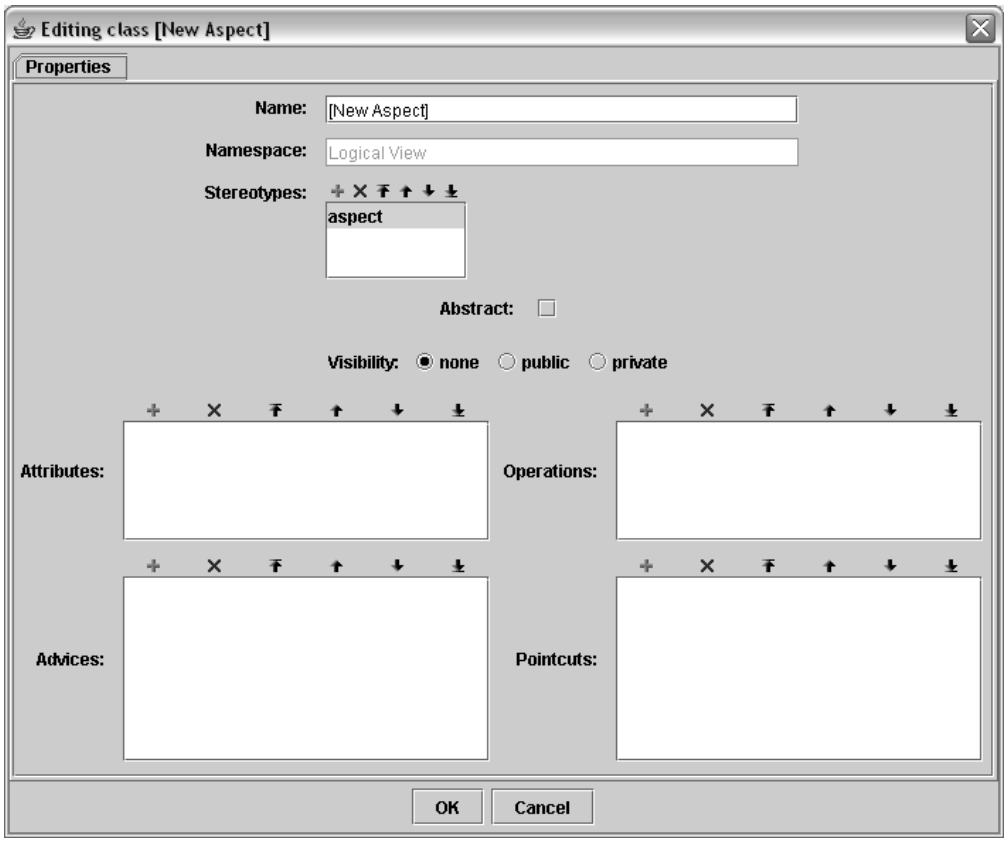


Figura 33: Especificação de um aspecto na MVCASE

Assim como uma classe, um aspecto também pode conter atributos (*Attributes*) e métodos (*Operations*), e, além deles, um aspecto pode conter adendos (*Advices*) e conjuntos de pontos de junção (*Pointcuts*).

A Figura 34 mostra o projeto orientado a aspectos parcial do Sistema de Caixa de Banco, relativo ao interesse de persistência em banco de dados.

Na Figura 34, os elementos relativos à POA estão sombreados. Em **(1)** tem-se o aspecto que encapsula o interesse de conexão com o banco, que por opção foi encapsulado separado do interesse de persistência, em **(2)**. Em **(3)** tem-se a representação dos dois métodos, declarados como intertipos, que serão adicionados à classe `Bankbox` no momento da combinação, o mesmo acontecendo para os métodos representados em **(4)**, que serão adicionados à classe `Client`. O aspecto de persistência relativo às contas do cliente **(5)**, separado desta forma por opção, também possui declarações intertipos. Em **(6)** tem-se o método que será inserido na classe `Account` e em (7) os que serão inseridos na classe `Bankbox`.

A Figura 35 mostra como os conjuntos de pontos de junção e os adendos são especificados na MVCASE.

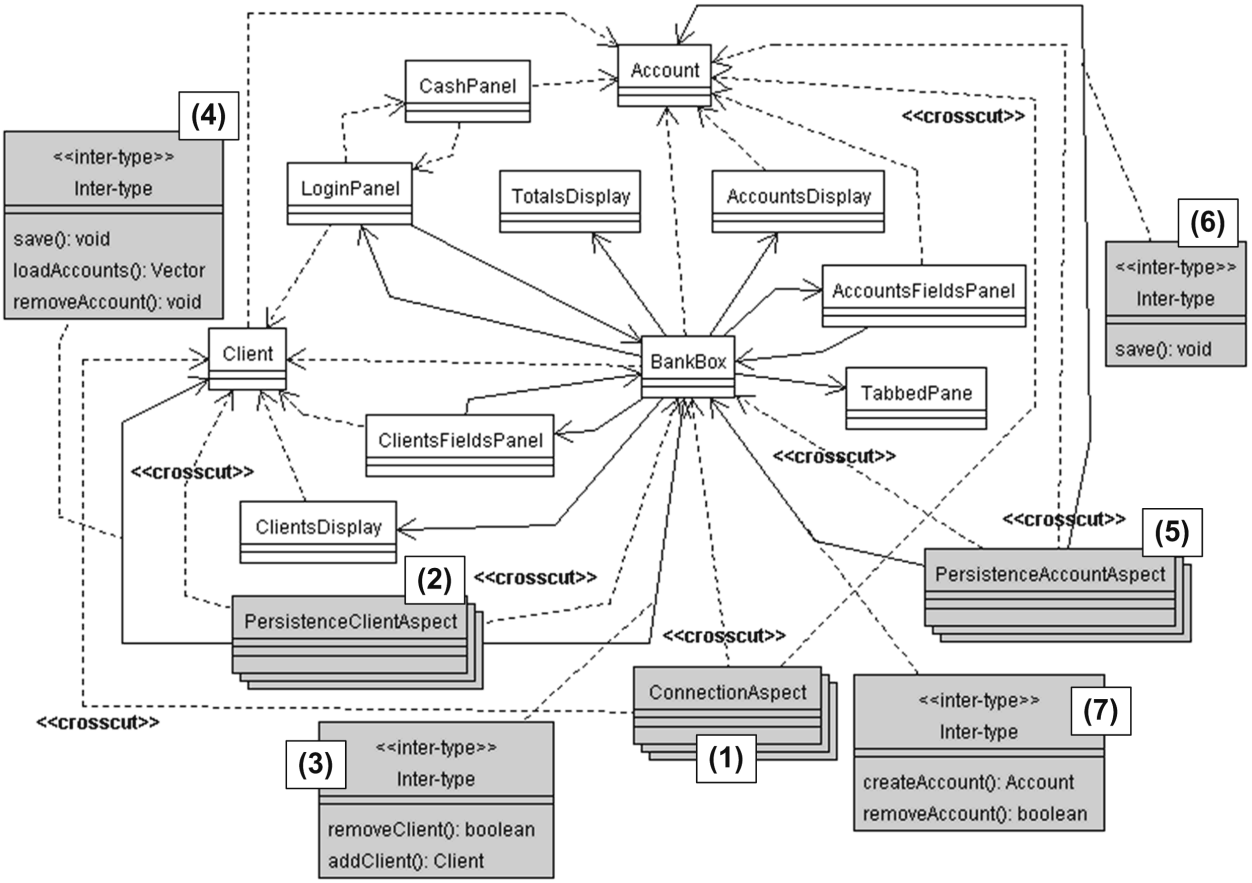


Figura 34: Projeto orientado a aspectos parcial do Sistema - Persistência e Conexão com banco

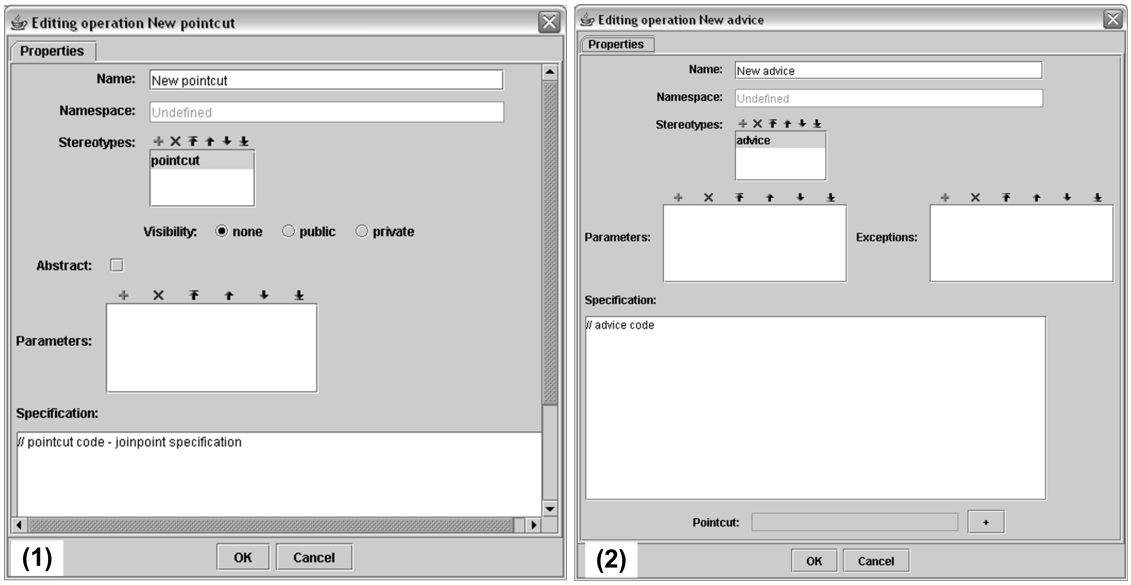


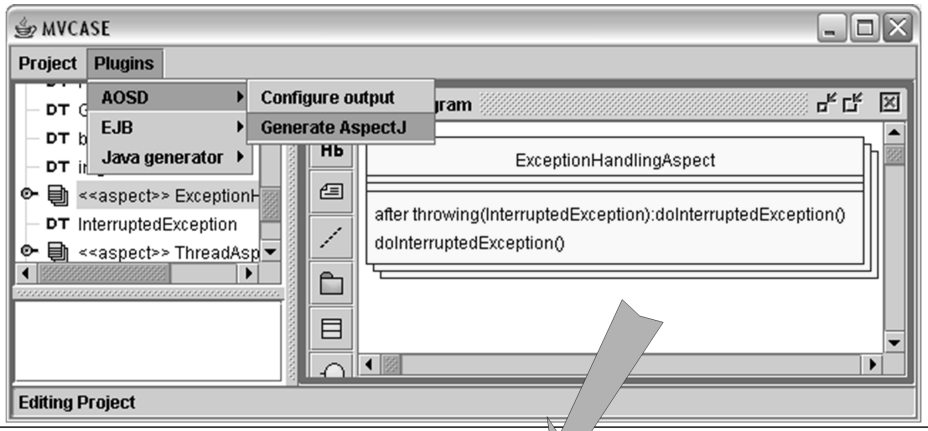
Figura 35: Especificação dos conjuntos de pontos de junção e adendos na MVCASE

Em **(1)** tem-se a tela para especificação dos conjuntos de pontos de junção. Os pontos de junção e os designadores são especificados como código dos conjuntos de pontos de junção (*"pointcut code - joinpoint specification"*). O mesmo acontece para o código dos adendos (*"advice code"*)

em **(2)**.

O projeto recuperado e o código orientado a aspectos obtido após a aplicação da abordagem constituem o conhecimento do sistema legado. Este conhecimento está encapsulado em uma documentação mais modularizada, com componentes para o reuso, atualizada e consistente. Utilizando este projeto obtido e importado na MVCASE, o Engenheiro de Software pode aplicar diferentes técnicas de modelagem para modificar requisitos funcionais e não-funcionais existentes, adicionar novas características ou definir a arquitetura física e lógica dos componentes do sistema.

Uma vez que o projeto orientado a aspectos está finalizado, o Engenheiro de Software pode implementá-lo utilizando uma linguagem orientada a aspectos. Essa tarefa pode ser realizada parcialmente pela MVCASE, por meio de um *plug-in* desenvolvido para gerar o código em *AspectJ*, a partir das descrições em XMI. Na Figura 36 é apresentada a geração do código do aspecto `ExceptionHandlingAspect`, na MVCASE.




```
public aspect ExceptionHandlingAspect {
 public pointcut doInterruptedException():
 execution (void Filosofo.Pensar()) || execution (void Filosofo.Comer()) ||
execution (public void Filosofo.setSleep(..)) || execution (synchronized int
Garfo.PegaGarfo()) || adviceexecution();
  declare soft: InterruptedException : doInterruptedException();
  after() throwing (InterruptedException e): doInterruptedException(){
    try{ throw new InterruptedException();
    } catch(InterruptedException e1){ e1.printStackTrace(); }
  }
}
```

Figura 36: Geração de código em *AspectJ* pela MVCASE

O código gerado é executado e testado. Se algum problema for identificado, o Engenheiro de Software pode retornar aos passos anteriores e corrigir o problema diretamente no código ou no projeto, já que o código pode ser gerado e testado mais uma vez. Esse processo é contínuo enquanto o sistema não passe nos testes. Um conjunto de testes de unidade e testes de cobertura podem ser criados para auxiliar na manutenção e na correção do sistema reconstruído.

Assim como as refatorações aplicadas para extrair os interesses transversais, a recupera-

ção do projeto orientado a aspectos pode ser auxiliada por transformações de software, semi-automatizando o processo e diminuindo a probabilidade de inserção de erros pelo Engenheiro de Software.

5 *Estudo de caso e avaliação da abordagem*

Neste capítulo é apresentada, na seção 5.1, a realização de um estudo de caso para ilustrar a aplicação da abordagem *Phoenix*. Em seguida, na seção 5.2, é feita uma avaliação dos resultados, com base nos dados colhidos na execução do estudo de caso. O sistema utilizado no estudo de caso foi obtido pela Internet.

5.1 *XSimulare*

O *XSimulare*[1] é um *framework* construído em Java para simulação de sistemas ambientais por meio da modelagem de organismos e seus relacionamentos entre si. A equipe de desenvolvimento foi formada por um grupo de nove estudantes (M.Sc. e Ph.D.) de Ciência da Computação da Universidade Federal de Pernambuco. O sistema possui 63 classes distribuídas em quatro pacotes e com um total de aproximadamente 7.5K linhas de código.

No *site* do sistema é possível ter acesso a um guia de usuário e a documentos da fase de desenvolvimento como, por exemplo, *Documento de Arquitetura*, *Documento de Requisitos* e o *Relatório de Testes*. Além disso, o código possui comentários. Neste contexto, o entendimento do sistema foi realizado por meio do estudo do guia do usuário e dúvidas técnicas foram sanadas por meio dos demais documentos.

A Figura 37 mostra a tela com o início de uma simulação.

Após o entendimento, o código do sistema foi analisado a fim identificar os possíveis interesses transversais, entrelaçados e dispersos através das classes do sistema. Neste estudo de caso foram identificados quatro interesses transversais relacionados a: tratamento de exceção, rastreamento, *logging* e atualização do relógio.

[1]Disponível em: `http://xsimulare.tigris.org/`.

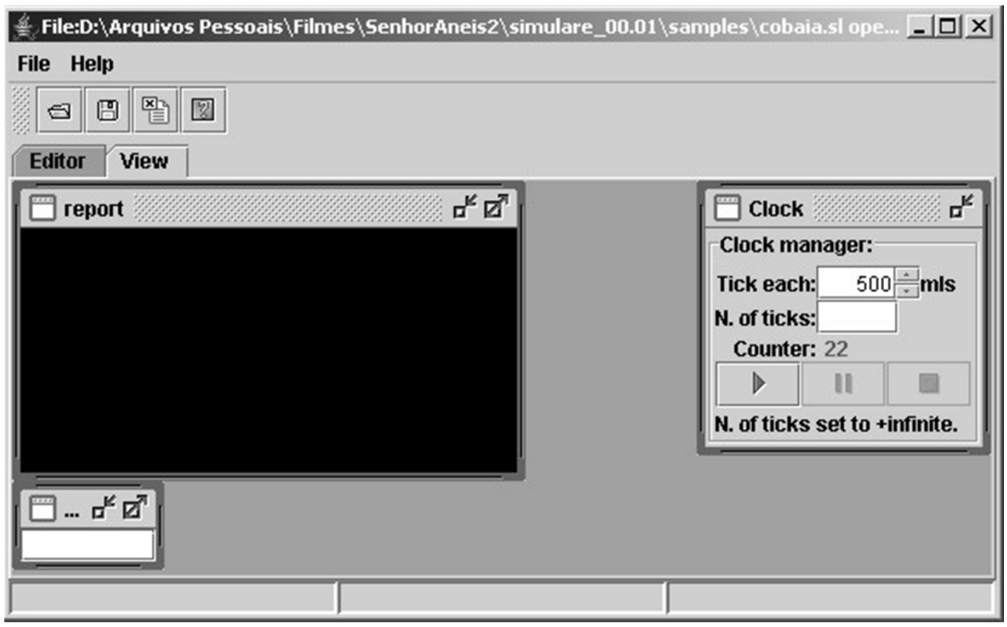


Figura 37: *XSimulare* - Início da simulação

5.1.1 Identificação dos Interesses Transversais

A identificação dos interesses, neste estudo de caso, ocorreu de duas maneiras:

i. Por **transformação de software**, dois interesses foram identificados: rastreamento e tratamento de exceção;

ii. Por **inspeção do código**, outros dois interesses foram identificados: *logging* e atualização do relógio da simulação.

A Figura 38 mostra alguns trechos de códigos relacionados com o interesse transversal de tratamento de exceção. Esta identificação, conforme descrita no capítulo 4, é baseada em *parser* e utiliza uma base de conhecimento para armazenar fatos contendo informações sobre variáveis, expressões, métodos e atributos analisados no código.

A Figura 39 mostra alguns fatos obtidos por meio das transformações que analisaram o código do sistema e armazenaram-nos na base de conhecimento. Maiores informações sobre a especificação dos fatos podem ser encontradas no Capítulo 4.

Na Figura 39, em **A** têm-se os fatos relativos à classe `Editor`, que estende a `JPanel` e implementa a `ClockListener`. Ela possui ainda o modificador `public` indicando que é pública. Em **B** têm-se os fatos relativos ao método `jbInit`, que é privado (modificador **private**), `void` e lança uma exceção do tipo `Exception` por meio da expressão `throws`. O fato `ExceptionHandlingCCByThrow` indica a presença do interesse transversal de tratamento de exceção pelo *throws* do método `jbInit`. Em **C**, têm-se os fatos relacionados ao método `openFile`,

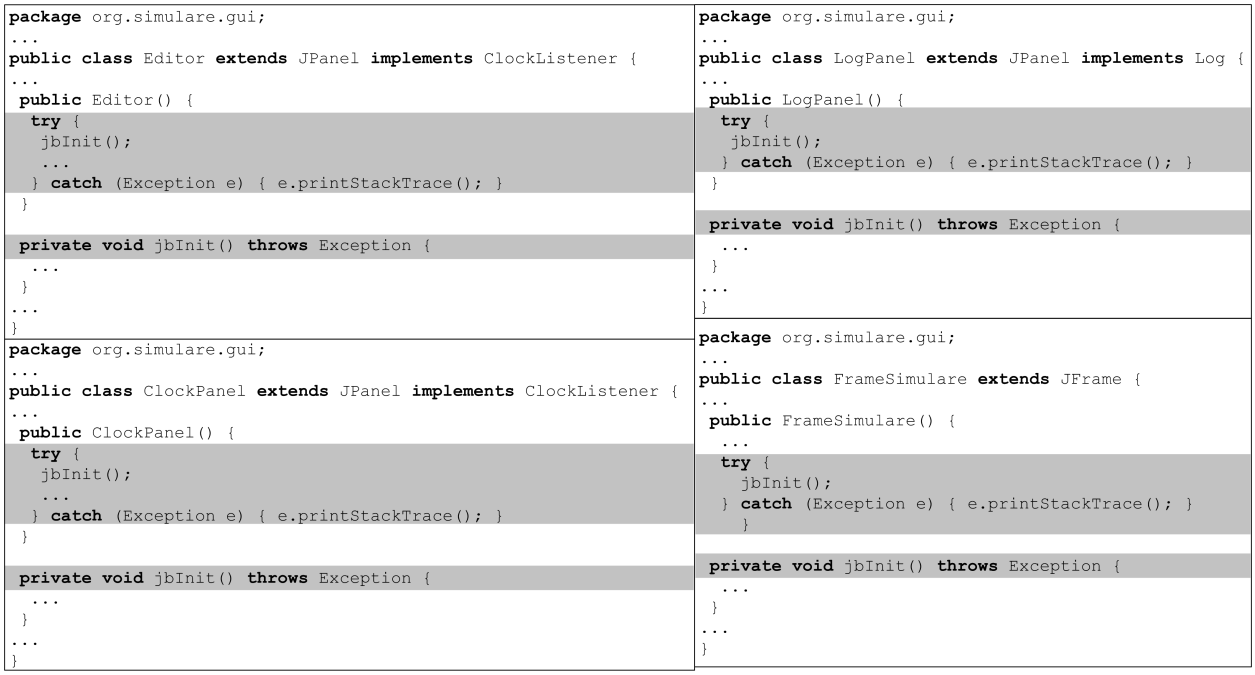


Figura 38: Identificando os interesses no código: Tratamento de Exceção

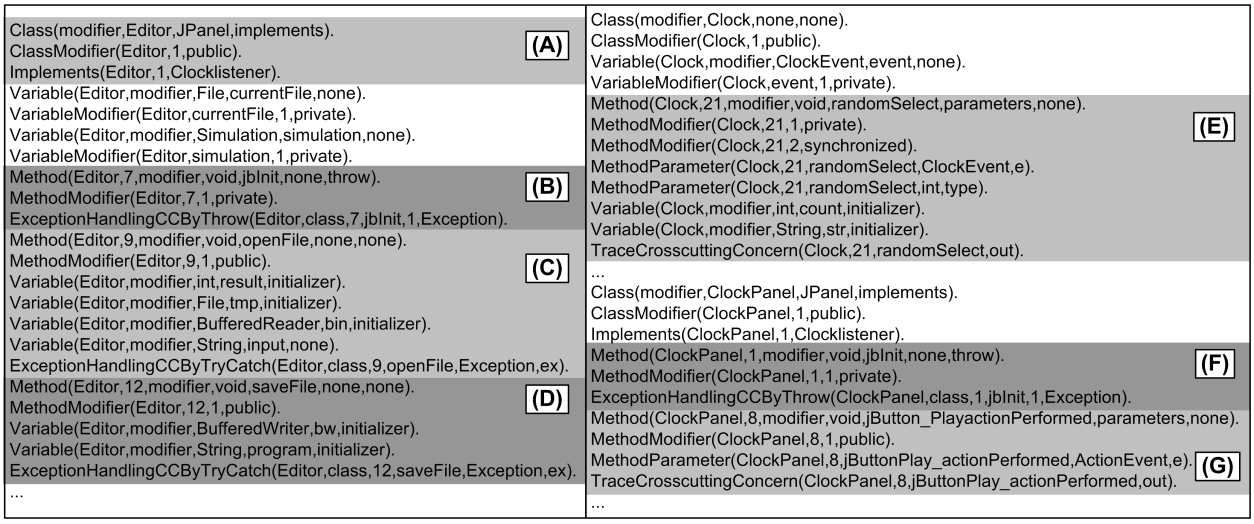


Figura 39: *XSimulare* - Visão parcial da Base de Conhecimento

que além de armazenar informações sobre características do método, também armazena informações sobre variáveis internas. Além disso, o fato `ExceptionHandlingCCByTryCatch(Editor, class, 9, openFile, Exception, ex)` indica a existência do interesse de tratamento de exceção pela presença da expressão *"try-catch"* no corpo do método, capturando uma exceção do tipo `Exception`. E em **D** têm-se os fatos relativos ao método `saveFile`, armazenando também informações sobre as características do método, variáveis internas e indicando a presença do interesse de tratamento de exceção pela existência de *"try-catch"*.

Ainda na Figura 39, têm-se em **E** fatos relacionados ao método `randomSelect`, da

classe `Clock`. Além das informações sobre as características do método, também são armazenadas na forma de fatos, informações sobre os parâmetros de entrada do método como, por exemplo, o fato `MethodParameter(Clock, 21, randomSelect, ClockEvent, e)`, que informa a existência de um parâmetro de entrada `e`, do tipo `ClockEvent`. Já o fato `TraceCrosscuttingConcern(Clock, 21, randomSelect, out)` indica a presença do interesse de rastreamento no método `randomSelect`, pela expressão *"System.out"* (parâmetro `out` no fato). Em **F** têm-se os fatos indicando a presença de tratamento de exceção, pelo lançamento de exceção (método com *throw*), na classe `ClockPanel`, método `jbInit`. E, finalmente, em **G**, têm-se os fatos relacionados ao interesse de rastreamento no método `jButtonPlay_actionPerformed`, por meio da expressão *"System.out"*.

Além do interesse relativo ao tratamento de exceções e ao rastreamento, também foi pesquisado por meio de transformações de software os interesses de: persistência em banco de dados e programação paralela, porém não foram encontrados no sistema *XSimulare*.

Após a identificação dos interesses por meio de transformação de software, partiu-se para a identificação dos interesses por meio da inspeção do código. A Figura 40 mostra os trechos de código relacionados ao interesse de atualização do relógio, identificados pela inspeção do código e pela documentação do sistema.

```
package org.simulare.gui;
import java.lang.reflect.*;
import java.util.*;
import org.simulare.lib.*;
/**
 * Represents a Task scheduled to execute in periods,
 * where the start and end time can be specified.
 */
public class Schedule extends ClockedTask {
 ...
 public void clockChanged(ClockEvent e) {
  if (isValid()) {
   // the clock can not have been stoped, if it does,
   // the even must be ignored.
   if (!e.stoped) {
    if (e.counter >= getStart()) {
     if (counter < repetition) {
      if (period == 0 ||
       ((e.counter - getStart()) % period) == 0) {
       counter++;
       Task t = getSimulation().getTask(getTask());
       setup(t);
       t.execute(getItem());
      }
     }
    }
   }
  }
 }
...
}
```

```
package org.simulare;
import java.util.*;
public class SimulationImpl implements Simulation {
 ...
 private Clock clock;
 ...
 public SimulationImpl() {
  clock = new Clock();
  ...
 }
 public Clock getClock() {  return clock; }
 public void setClock(Clock clock) { this.clock = clock; }
 private void remove(Task task, Item item) {
 List list = new LinkedList();
 Iterator iter = schedule.iterator();
 while (iter.hasNext()) {
  ClockedTask ct = (ClockedTask) iter.next();
  if (task != null && ct.getTask().equals(task.getName())) {
   list.add(ct);
  }
  if (item != null && ct.getItem().equals(item.getName())) {
   list.add(ct);
  }
 }
 iter = list.iterator();
 while (iter.hasNext()) {
  removeSchedule((Schedule) iter.next());
 }
}
...
}
```

Figura 40: Identificando os interesses no código: Atualização do Relógio

O passo *Identificação de Interesses Transversais* deve ser realizado até que todos os interesses sejam identificados pelo Engenheiro de Software, ou até que ele decida que já identificou os

interesses necessários para o seu processo.

Finalizada a identificação, partiu-se para a aplicação das refatorações, visando obter o sistema reorganizado segundo os princípios da Orientação a Aspectos. As informações obtidas pelas transformações de software, bem como as obtidas pela inspeção do código foram utilizadas no próximo passo.

5.1.2 Reorganização Aspectual

As refatorações apresentadas na Seção 4.2.2, Capítulo 4, foram utilizadas para extrair os interesses e encapsulá-los em aspectos.

Considere o trecho de código da classe `SimulationImpl`, relacionado ao interesse de atualização do relógio da simulação.

```
    package org.simulare;
    import java.util.*;
    public class SimulationImpl implements Simulation {
      // simulation controler
(1)   private Clock clock;
      ...
(2)   public Clock getClock() {
       return clock;
      }
(3)   public void setClock(Clock clock) {
       this.clock = clock;
      }
      private void remove(Task task, Item item) {
        List list = new LinkedList();
        Iterator iter = schedule.iterator();
        while (iter.hasNext()) {
          ClockedTask ct = (ClockedTask) iter.next();
          if (task != null && ct.getTask().equals(task.getName())) {
            list.add(ct);
          }
          if (item != null && ct.getItem().equals(item.getName())) {
            list.add(ct);
          }
        }
        iter = list.iterator();
        while (iter.hasNext()) {
          removeSchedule((Schedule) iter.next());
        }
      }
      ...
    }
```

Por meio da aplicação das refatorações, o interesse de atualização do relógio da simulação foi extraído e encapsulado em um aspecto. A primeira refatoração aplicada é a **EXTRACT FIELD TO ASPECT**, para extrair o atributo `clock` do tipo `Clock`, sombreado em **(1)**. Em seguida, aplicou-se a **EXTRACT METHOD TO ASPECT** para extrair os métodos `getClock` (sombreado em **(2)**) e `setClock` (sombreado em **(3)**) relacionados ao interesse de atualização do relógio da simulação para o aspecto. Após as refatorações serem aplicadas, tem-se:

```
package org.simulare;
import java.util.*;
public aspect Foo {
  //simulation controler
public org.simulare.Clock org.simulare.SimulationImpl.clock;
  public org.simulare.Clock org.simulare.SimulationImpl.getClock(){
    return clock;
  }
  public void org.simulare.SimulationImpl.setClock(org.simulare.Clock clock) {
    this.clock = clock;
  }
}
```

Em seguida, foi identificado que o código do corpo do método `remove` relacionava-se com o interesse de atualização do relógio da simulação, que estava sendo extraído para um aspecto. Porém, a assinatura do método não estava relacionada ao contexto do interesse. Aplicou-se então a refatoração **EXTRACT CODE TO ADVICE** para mover o código do corpo do método para um adendo, mantendo a assinatura do método na classe `SimulationImpl`, obtendo-se:

```
package org.simulare;
import java.util.*;
public class SimulationImpl implements Simulation {
  ...
  private void remove(Task task, Item item) { }
  ...
}
```

E no aspecto, o adendo:

```
void around(org.simulare.SimulationImpl si, Task task, Item item):
execution (private void org.simulare.SimulationImpl.remove(org.simulare.Task,
org.simulare.Item)) && target(si) && args(task, item);{
  List list = new LinkedList();
  Iterator iter = si.getSchedule().iterator();
  while (iter.hasNext()) {
    ClockedTask ct = (ClockedTask) iter.next();
    if (task != null && ct.getTask().equals(task.getName())) {
      list.add(ct);
```

```
    }
    if (item != null && ct.getItem().equals(item.getName())) {
      list.add(ct);
    }
  }
  iter = list.iterator();
  while (iter.hasNext()) {
    si.removeSchedule((Schedule) iter.next());
  }
}
```

Como todo o corpo do método foi movido para o adendo, este teve que ser um adendo de contorno, que toma para si o controle da computação quando o método for executado (designador `execution`). Em um adendo de contorno, o comportamento especificado é executado **EM LUGAR** do comportamento original do método afetado, que não é executado. Nota-se que as variáveis passadas como parâmetro para o adendo tem a função de capturar o contexto da execução do método.

Após a extração do corpo do método para um adendo no aspecto, observou-se que a assinatura do adendo tornou-se complexa. A fim de melhorar a legibilidade, aplicou-se a refatoração **EXTRACT POINTCUT DEFINITION**. Em seguida, para fornecer um nome significativo ao conjunto de pontos de junção, aplicou-se a **RENAME POINTCUT**. O seguinte conjunto de pontos de junção foi criado, e uma chamada a ele foi inserida no adendo (em sombreado):

```
public pointcut doClockSimulationImplRemove(org.simulare.SimulationImpl si,
org.simulare.Task task, org.simulare.Item item): execution (private void
org.simulare.SimulationImpl.remove(org.simulare.Task, org.simulare.Item)) &&
target(si) && args(task, item);

void around(org.simulare.SimulationImpl si, Task task, Item item):
doClockSimulationImplRemove(si, task, item) {
  List list = new LinkedList();
  Iterator iter = si.getSchedule().iterator();
  while (iter.hasNext()) {
    ClockedTask ct = (ClockedTask) iter.next();
    if (task != null && ct.getTask().equals(task.getName())) {
      list.add(ct);
    }
    if (item != null && ct.getItem().equals(item.getName())) {
      list.add(ct);
    }
  }
  iter = list.iterator();
  while (iter.hasNext()) {
    si.removeSchedule((Schedule) iter.next());
  }
}
```

Na execução deste estudo de caso, foram encontrados trechos de código relacionados a interesses transversais que deveriam ser encapsulados em aspectos como, por exemplo, quando uma classe implementa uma interface ou estende uma outra classe. Um exemplo dito pode ser visto na Figura 40, onde a classe `Schedule` estende a classe `ClockedTask`. Porém não haviam sido especificadas refatorações para esta situação em especial. Neste caso, a solução encontrada foi extrair para o aspecto essa propriedade como uma declaração intertipos, conforme se segue:

```
declare parents: org.simulare.Schedule extends org.simulare.ClockedTask;
```

Finalmente, após todo interesse transversal de atualização do relógio estar encapsulado no aspecto `Foo`, foi aplicada a refatoração **RENAME ASPECT** para lhe dar um nome mais significativo. Como resultado final da extração do interesse transversal de atualização do relógio foi obtido o seguinte aspecto, conforme mostra a Figura 41.

```
package org.simulare;
import java.util.*;
import javax.swing.*;
import org.simulare.gui.*;
public aspect ClockAspect {
  declare parents: Schedule extends ClockedTask;
  public void Schedule.clockChanged(ClockEvent e) {
    if (isValid()) {
    // the clock can not have been stoped, if it does,
the even must be ignored.
      if (!e.stoped) {
        if (e.counter >= getStart()) {
          if (counter < getRepetition()) {
            if (getPeriod() == 0 || ((e.counter -
getStart()) % getPeriod()) == 0) {
              counter++;
              Task t =
getSimulation().getTask(getTask());
              setup(t);
              t.execute(getItem());
            }
          }
        }
      }
    }
  }
  //simulation controler
  public Clock SimulationImpl.clock;
  public Clock SimulationImpl.getClock(){
    return clock; }
  public void SimulationImpl.setClock(Clock clock) {
    this.clock = clock; }
  declare parents: ClockPanel implements
ClockListener;
  public Clock ClockPanel.clock;
  public JSpinner ClockPanel.jSpinnerClock = new
JSpinner();
  public Clock ClockPanel.getClock() { return clock; }
  public void ClockPanel.setClock(Clock clock) {
    if (this.clock != null) {
      this.clock.removeClockListener(this); }
    this.clock = clock;
    this.clock.addClockListener(this);
    jSpinnerClock.setValue(new
Long(clock.getPeriod()));
  }
  declare parents: Editor implements ClockListener;
  public pointcut doClockSimulationImpl(SimulationImpl
si):
  execution (public SimulationImpl.new(..)) &&
target(si);
  after(SimulationImpl si): doClockSimulationImpl(si){
si.setClock(new Clock());}
  public pointcut
doClockSimulationImplRemove(SimulationImpl si, Task
task, Item item): execution (private void
SimulationImpl.remove(Task, Item))
  && target(si) && args(task, item);
  void around(SimulationImpl si, Task task, Item
item): doClockSimulationImplRemove(si, task, item){
    List list = new LinkedList();
    Iterator iter = si.getSchedule().iterator();
    while (iter.hasNext()) {
      ClockedTask ct = (ClockedTask) iter.next();
      if (task != null &&
ct.getTask().equals(task.getName())) { list.add(ct); }
      if (item != null &&
ct.getItem().equals(item.getName())) { list.add(ct); }
    }
    iter = list.iterator();
    while (iter.hasNext()) {
      si.removeSchedule((Schedule) iter.next()); }
  }
  public pointcut
doClockClockPanelConstructor(ClockPanel cp):
  call (public ClockPanel.new(..)) && target(cp);
  after(ClockPanel cp):
doClockClockPanelConstructor(cp){
    cp.setClock(new Clock());
  }
  public pointcut
doClockClockPaneljSpinnerClock(ClockPanel cp):
  call (void
ClockPanel.jSpinnerClock_stateChanged(..)) &&
target(cp);
  after(ClockPanel cp):
doClockClockPaneljSpinnerClock(cp){
    Number n = (Number) cp.jSpinnerClock.getValue();
    cp.clock.setPeriod(n.longValue());
    cp.jLabelStatus.setText("Tick set to "+n+" mls.");
  }
}
```

Figura 41: ClockAspect

Assim como a *Identificação de Interesses Transversais*, o passo *Reorganização Aspectual* deve ser realizado até que todos os interesses sejam extraídos e encapsulados em aspectos, ou até que Engenheiro de Software decida que já extraiu os interesses necessários para o seu processo.

Após todo o código estar orientado a aspectos, as transformações obtêm o projeto orientado a aspectos do sistema, conforme pôde ser visto na Figura 31 (capítulo 4). Este último passo é opcional, ficando a cargo do Engenheiro de Software decidir sobre a sua realização ou não. De posse do projeto orientado a aspectos, futuramente, possíveis modificações poderão ser feitas no próprio modelo, em um alto nível de abstração, e em seguida por meio da próprio ferramenta MVCASE o código pode ser gerado novamente em *AspectJ*.

5.1.3 Recuperação do Projeto Orientado a Aspectos

A Figura 42 mostra as descrições na linguagem de modelagem XMI para o aspecto `ClockAspect`, obtidas por meio da transformação de software `AspectJ2XMI` implementada no Draco-PUC.

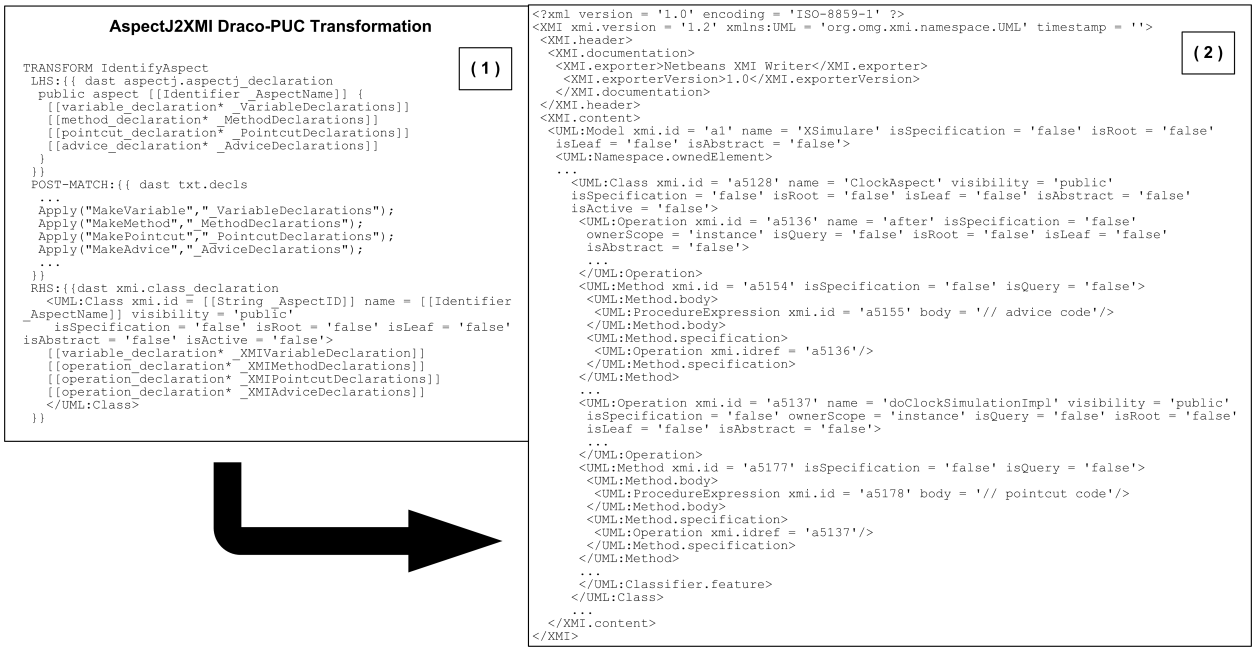


Figura 42: Transformação AspectJ2XMI

Em **(1)** tem-se o trecho de código do transformador `IdentifyAspect`, responsável por identificar a declaração de um aspecto público, e mapear as suas descrições em AspectJ (regra LHS) para descrições em XMI (regra RHS). Em **(2)** têm-se as descrições XMI para o aspecto `ClockAspect`, cujo trecho de código foi visto na seção anterior.

A Figura 43 mostra o projeto orientado a aspectos parcial do Sistema *XSimulare*, relativo ao interesse de atualização do relógio. Os elementos sombreados são relativos à POA. O aspecto `ClockAspect` encapsula o interesse de atualização do relógio. Por meio do relacionamento *<<declare parents>>* **(1)** o aspecto está alterando dinamicamente a hierarquia das classes, informando que após a combinação a classe `Schedule` passa a estender a classe

`ClockedTask`. O relacionamento de entrecorte (<<*crosscut*>>) em **(2)** informa que o aspecto `ClockAspect` entrecorta a classe `ClockPanel` por meio do conjunto de pontos de junção `doClockClockPanelConstructor` e, em **(3)**, tem-se a declaração deste conjunto de pontos de junção, bem como a declaração de um adendo do tipo `after`, relacionado a este conjunto de pontos de junção.

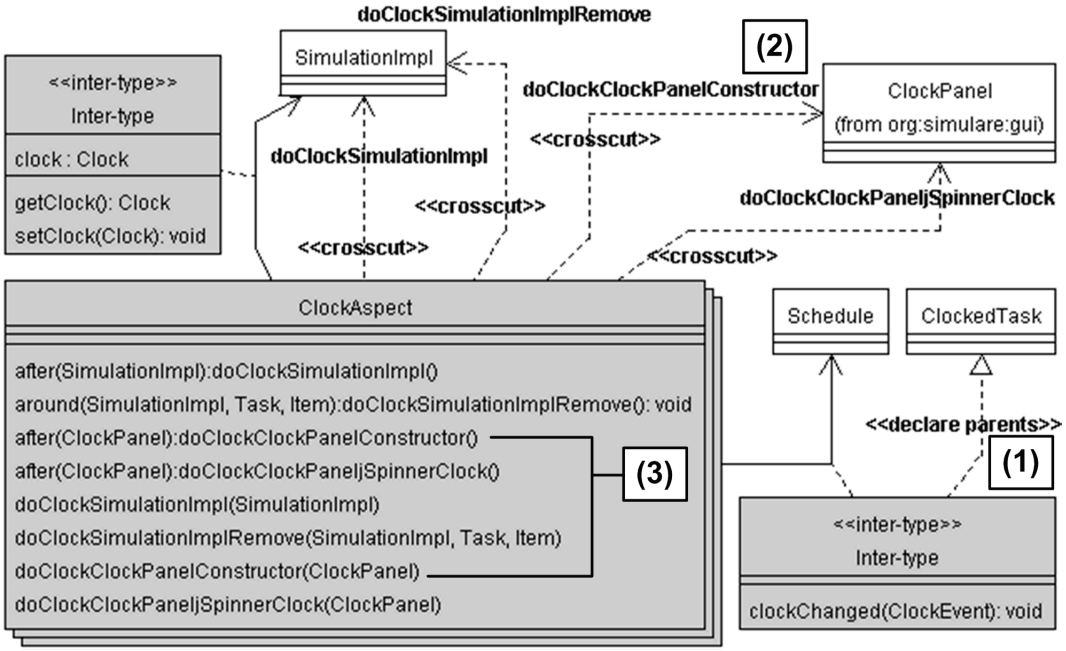


Figura 43: Projeto orientado a aspectos parcial do sistema: Atualização do relógio

5.2 Discussão

Além do estudo de caso apresentado neste capítulo, para avaliação da abordagem, outros 3 estudos de caso, de diferentes domínios, foram realizados. Estes outros 3 estudos de caso foram realizados com o objetivo de auxiliar na definição da abordagem, mais especificamente na especificação das expressões regulares, e posteriormente na implementação delas nas transformações de software, para a identificação dos interesses transversais.

Após a realização dos estudos de caso, pôde-se comprovar uma redução no número de linhas de código dos sistemas. Para o sistema de Caixa de Banco a redução foi de 168 linhas de código correspondendo a 6,74%. No Sistema de Vendas pela Web a redução foi de 112 linhas de código, correspondendo a 5,63%. Já no sistema do Jantar dos Filósofos foi de 46 linhas de código, correspondendo a 10,82%. E por fim, no *XSimulare* a redução foi de 307 linhas de código, correspondendo a 4,14%. A Tabela 4 mostra os resultados obtidos nos 4 estudos de caso.

O número de linhas de código reduzidas e o aumento no número dos módulos (aspectos)

Estudos de caso	Orientado a Objetos			Orientado a Aspectos				Dif.
	Pacotes	Classes	LOC	Pacotes	Classes	Aspectos	LOC	(%)
Caixa de Banco	-	11	2492	-	11	6	2324	6,74
Vendas pela Web	-	12	1989	-	12	3	1877	5,63
Filósofos	-	4	425	-	4	2	379	10,82
XSimulare	4	63	7423	4	61	4	7116	4,14

Tabela 4: Avaliação dos Estudos de Caso

indicam que os elementos (classes e aspectos) recuperados são menores e mais bem divididos, já que cada aspecto agrupa um único interesse (menor entrelaçamento de código). Pode-se afirmar que os módulos são também mais coesos, devido ao fato de que o espalhamento de código é reduzido pela utilização do DSOA.

A partir da análise do código e do projeto orientado a aspectos recuperados pela aplicação da abordagem *Phoenix*, pode-se inferir que as conseqüências em se identificar e separar os interesses transversais são equivalentes aos estabelecidos pelo DSOA.

Como benefícios pode-se destacar:

i. Automatização: O uso de transformações de software, com o Draco-PUC, automatiza um estágio importante no reengenharia: a recuperação do projeto do sistema. As atividades da engenharia reversa, tais como a identificação dos interesses transversais e a reorganização aspectual, também são aceleradas devido à esta automatização;

ii. Rastreamento de requisitos : Após a separação dos interesses, fica mais fácil rastrear cada módulo a um requisito específico;

iii. Melhoria na Manutenção: As mudanças de requisitos, melhorias da funcionalidade e reestruturação de código são também mais fáceis de executar, devido principalmente à melhor divisão dos módulos e maior coesão;

iv. Legibilidade: O novo código é mais claro e menos poluído, porque as operações com os atributos e os métodos não são espalhadas e entrelaçadas através do sistema;

v. Reuso: Os aspectos identificados e extraídos podem ser implementados de tal maneira que podem ser reutilizados futuramente. Isto pode dar origem, com a realização de diferentes estudos de caso, a um *framework* de aspectos para domínios de interesses transversais específicos, como, por exemplo, rastreamento, tratamento de exceção e persistência em banco de dados.

Além disso, algumas desvantagens puderam ser observadas:

Construção dos Transformadores: Visto que a abordagem *Phoenix* usa um sistema transformacional para auxiliar o Engenheiro de Software em algumas tarefas do processo, admite-se

que devem existir os transformadores para identificar os interesses transversais, para reorganizar o código por meio da aplicação de algumas refatorações e para recuperar as informações de projeto diretamente do código. Entretanto, estes transformadores devem primeiramente ser construídos, o que requer um esforço e um grande conhecimento sobre as linguagens envolvidas, visto que além das linguagens do próprio sistema transformacional para especificar as gramáticas e os transformadores, também é necessário ter um domínio sobre as linguagens de programação e de modelagem (*Java*, *AspectJ* e *XMI*, por exemplo). O tempo gasto na construção dos transformadores também é um fator crítico. Porém, deve-se reforçar que este esforço é reutilizado futuramente em todo sistema escrito na mesma linguagem, como por exemplo, os transformadores construídos para identificação de interesses transversais em sistemas *Java* é genérico para todo sistema implementado em *Java*, por se basear na sua *BNF*. A redução do tempo obtida quando se utilizar estes transformadores na recuperação do projeto, por exemplo, justifica este esforço inicial; e

Notação orientada a aspectos: A notação UAE especificada para representação de um sistema orientado a aspectos em alto nível de abstração, apesar de ter sido definida visando ser genérica para representar os conceitos do DSOA, possui algumas particularidades da linguagem AspectJ. Essas particularidades tiveram de ser inseridas para possibilitar a geração de código da maneira mais próxima possível do que deve ser o produto final: o código executável orientado a aspectos. Assim, surge a necessidade de avaliar a utilização da ferramenta *MVCASE*, com a notação UAE, na especificação de sistemas orientados a aspectos em outras linguagens como, por exemplo, *AspectC++* (ASPECTC++, 2004). Além disso, algumas características particulares da linguagem *AspectJ* não são expressas como, por exemplo, as expressões *"declare soft"* e *"declare warning"* e aspectos privilegiados.

Testes: Não existe na abordagem *Phoenix* uma metodologia bem definida de testes para o sistema orientado a aspectos obtido. A verificação é feita de forma *ad-hoc* pelo Engenheiro de Software. Porém, testes de regressão, por exemplo, poderiam ser aplicados para verificar se a funcionalidade do sistema orientado a objetos foi preservada no sistema orientado a aspectos obtido (RAMOS, 2004).

6 *Considerações finais e trabalhos futuros*

Esta dissertação apresenta a abordagem *Phoenix* para a Reengenharia de Software Orientada a Aspectos. Diversas tecnologias foram estudadas e aplicadas, como Mineração de Aspectos, Refatorações e Desenvolvimento de Software Orientados a Aspectos, integradas com técnicas de Engenharia Reversa e Reengenharia.

6.1 Trabalhos correlatos

O primeiro trabalho relevante que envolve a tecnologia da Orientação a Objetos e a recuperação do conhecimento embutido em sistema legados foi apresentado por Jacobson e Lindstron (JACOBSON; LINDSTROM, 1991), que aplicaram a reengenharia nos sistemas legados implementados em linguagens procedurais. Os autores indicam que a reengenharia deve ser realizada de uma maneira gradual, porque seria impraticável substituir completamente um sistema antigo por um novo.

Como foi visto na Seção 2.2.4, do Capítulo 2, uma nova camada, baseada em componentes, está sendo incorporada ao processo de desenvolvimento de software. Os objetivos desta camada são similares aos da Orientação a Objetos, buscando um maior reuso para aumentar a confiabilidade, segurança e diminuir os custos do software (LEE et al., 2003).

Ainda na Seção 2.2.4 foi apresentada a pesquisa de Caldiera e Basili (CALDIERA; BASILI, 1991) como sendo um dos primeiros trabalhos relevantes neste sentido. Os autores buscavam a extração automatizada de componentes de software reutilizáveis de sistemas existentes. O processo proposto é dividido em duas fases distintas onde, primeiramente, são escolhidos os candidatos a componentes e, em seguida, esses são empacotados de forma a possibilitar o seu uso independente. Na segunda fase, um engenheiro com conhecimento do domínio da aplicação analisa esses candidatos a componentes para poder determinar quais serviços cada um deles pode fornecer.

Nesta era pós-Orientação a Objetos, as pesquisas sobre Orientação a Aspectos direcionam-se para determinar onde esta nova tecnologia pode ser utilizada visando melhorar o processo de

desenvolvimento e de manutenção de software, conforme discutido por Bayer (2000). A Orientação a Aspectos pode reduzir a complexidade e o entrelaçamento de código, aumentando assim a modularidade e o grau de reuso, problemas esses normalmente enfrentados pelos processos de Reengenharia de Software.

Em um processo de desenvolvimento de software, diversas atividades estão envolvidas, desde a especificação dos requisitos, análise, projeto, até a implementação em um código executável. Neste contexto, algumas pesquisas buscam utilizar as idéias do DSOA em conjunto com processos de reengenharia.

Os primeiros trabalhos nesta direção foram os realizados por Kendall (2000) e Lippert e Lopes (2000), ambos já discutidos na seção 2.2.5. Porém, o primeiro processo de reengenharia de software orientado a aspectos sistemático foi apresentado por Ramos (RAMOS, 2004).

Ramos propôs uma abordagem de reengenharia denominada *Aspectig* que busca a elicitação de aspectos em sistemas orientados a objetos, com a sua posterior implementação de acordo com o paradigma orientado a aspectos. A abordagem é realizada em três etapas distintas, que compreendem: Entender a Funcionalidade do Sistema, Tratar Interesses e Comparar o Sistema Orientado a Aspectos com o Orientado a Objetos. Diretrizes auxiliam a modelagem dos interesses identificados, utilizando diagrama de classes com a notação UML. Outras diretrizes auxiliam a implementação desses interesses em aspectos, utilizando a linguagem *AspectJ*. A utilização de testes de regressão no sistema após a reorganização do código do sistema agora implementado com aspectos, verificando que a funcionalidade original foi preservada, completa a abordagem.

A principal diferença existente entre a *Phoenix* e a *Aspecting* é que a primeira possui ferramentas que automatizam as principais tarefas da Reengenharia, garantindo assim, uma maior confiabilidade no resultado. Em compensação, a abordagem *Aspecting* possui uma etapa para verificação e validação do resultado baseada em testes de regressão, etapa esta que falta à abordagem *Phoenix* e pode ser considerado como um dos seus pontos fracos.

6.2 Contribuições

A principal contribuição deste trabalho é a definição de uma abordagem, denominada *Phoenix*, que apóia a reconstrução de sistemas legados por meio da sua reengenharia. A abordagem é baseada no paradigma orientado a aspectos, e é utilizada na obtenção de um novo sistema orientado a aspectos, mais modular e de mais fácil manutenção.

No contexto da identificação de interesses transversais, as transformações baseadas em um *parser* para realizar a mineração de aspectos pode ser considerada como contribuição, assim como

a especificação da gramática da linguagem Java no Draco-PUC, que teve de sofrer modificações para atender ao objetivo proposto nesta dissertação; e a gramática da linguagem *AspectJ*, ambas especificadas no Draco-PUC.

Esta pesquisa também contribuiu com a especificação de um catálogo de refatorações (GARCIA et al., 2004a, 2004) para auxiliar o Engenheiro de Software na extração de interesses transversais do código orientado a objetos. As transformações implementadas para a extração automática dos trechos de código relacionados com o interesse de tratamento de exceções também são uma importante contribuição.

Há também a extensão da UML para representação de aspectos em um alto nível de abstração, permitindo que o Engenheiro de Software separe interesses transversais nos estágios iniciais do desenvolvimento de software (GARCIA et al., 2004b). A ferramenta MVCASE foi então estendida para apoiar o DSOA por meio dessa notação e do mecanismo de geração automática de código em *AspectJ*.

Outra contribuição desta pesquisa vem da integração de diferentes técnicas e mecanismos para apoiar a abordagem *Phoenix*, automatizando parte das tarefas do Engenheiro de Software.

Por fim também pode ser considerada como contribuição desta pesquisa o uso combinado de uma ferramenta CASE com a ferramenta Draco-PUC, para apoiar a reengenharia em diferentes níveis de abstração, que vão desde o código do sistema orientado a objetos, até o seu projeto orientado a aspectos.

6.3 Trabalhos futuros

Outra forma de contribuição importante em um trabalho de pesquisa é a geração de trabalhos futuros. Com relação à presente dissertação, uma série de possibilidades foi identificada.

A mineração de aspectos na abordagem *Phoenix* é realizada por meio de transformações de software, sendo portanto baseada em *parser*, permitindo também a inserção da análise de seqüência de caracteres e a utilização de expressões regulares. Porém, inserir novas análises e/ou expressões regulares, bem como modificá-las, é uma tarefa árdua para o Engenheiro de Software, que terá de realizar a manutenção nos transformadores, de forma que com o passar do tempo a complexidade das regras de transformação o tornem muito confuso. Uma solução seria explorar a construção de uma ferramenta que apoiasse a mineração baseada em *parser*, mas sem a utilização de transformadores, para os interesses transversais de domínios já conhecidos (tratamento de exceções, persistência, rastreamento, entre outros), e que permitisse também ao Engenheiro de Software construir *"dinamicamente"* suas métricas de análise e expressões regulares.

Ainda considerando a construção de uma ferramenta para mineração, uma visualização gráfica dos possíveis interesses transversais do código pode ser desenvolvida. Dessa forma, a tarefa de identificação dos diferentes interesses no sistema será facilitada.

No contexto do catálogo de refatorações, elas são um primeiro passo em direção a um catálogo mais completo e que possa atender, senão a todas, à grande maioria dos problemas de extração de interesses. Também devem ser especificadas refatorações para trabalhar somente em código orientado a aspectos, permitindo a manutenção e a melhoria na legibilidade dos aspectos e seus elementos (conjuntos de pontos de junção, declaração intertipos, adendos) e na própria hierarquia de um sistema orientado a aspectos.

A aplicação das refatorações é um ponto que também pode derivar outras pesquisas, principalmente no que diz respeito à automatizar ou semi-automatizar essa aplicação. Diversas pesquisas avançam nesse sentido para as refatorações orientadas a objetos e tudo indica que possam também ser direcionadas para as refatorações orientadas a aspectos, como, por exemplo, o trabalho de Mendonça e outros (MENDONÇA et al., 2004).

A abordagem *Phoenix* carece de uma metodologia de testes para garantir que o resultado do processo está de acordo com o esperado. Porém, como a tecnologia orientada a aspectos é muito nova, ainda não existem métodos de teste bem definidos para os sistemas orientados a aspectos. Trabalhos nesse sentido vêm sendo desenvolvidos (BRUEL et al., 2003; HUGHES; GREENWOOD, 2003; MONK; HALL, 2002).

Em relação à ferramenta MVCASE e a notação UAE, ambas devem ser utilizadas em diferentes estudos de caso para validá-las e comprovar a sua eficácia. Novos diagramas podem ser desenvolvidos para auxiliar na documentação e no DSOA como, por exemplo, um diagrama que represente como serão os aspectos, classes, e seus relacionamentos após a combinação. Outro diagrama muito útil seria para representar o fluxo de execução do sistema orientado a aspectos (uma extensão do Diagrama de Seqüência).

Um problema relacionado à notação UAE é a sua natureza particularizada à linguagem *AspectJ*. Uma solução seria utilizar uma notação genérica (CHAVEZ, 2004) e uma específica, transformando de uma para outra. Esta é a idéia da abordagem MDA (*Model-Driven Architecture* ou Desenvolvimento Orientado a Modelos) (KLEPPE et al., 2003) da OMG. A utilização da MDA na abordagem *Phoenix* permitiria uma rapidez no desenvolvimento, maior e melhor manutenção, flexibilidade, reutilização e distribuição da aplicação orientada a aspectos. Proposta pelo OMG, a MDA busca aumentar o nível de abstração no desenvolvimento, dando maior importância aos modelos. É uma idéia recente, sendo ainda bastante discutida e pouco consolidada. A idéia é que, por meio da MVCASE, o Engenheiro de Software possa gerar aplicações completas, trabalhando

diretamente em um modelo visual e usar a sincronização ativa para manter o modelo e o código atualizados durante mudanças rápidas na aplicação.

Referências

ABRAHÃO, S.; PRADO, A. Web-enabling legacy systems through software transformations. In: *Proceedings of IEEE International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*. [S.l.: s.n.], 1999. p. 149–152.

ALMEIDA, E. et al. MVCASE: An integrating technologies tool for distributed component-based software development. In: *Proceedings of the 6th Asia-Pacific Network Operations and Management Symposium. (APNOMS'2002) Poster Session*. [S.l.]: IEEE Computer Society Press, 2002a.

ALMEIDA, E. et al. Ferramenta MVCASE - uma ferramenta integradora de tecnologias para o desenvolvimento de componentes distribuídos. In: *Simpósio Brasileiro de Engenharia de Software (SBES'2002). Sessão de Ferramentas*. [S.l.: s.n.], 2002b.

ALVARO, A. et al. Orion-RE: A Component-Based Software Reengineering Environment. In: *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*. [S.l.]: IEEE Computer Society Press, 2003. p. 248–257.

ARGOUML. *ArgoUML tool*. 2003. Disponível em: <http://argouml.tigris.org>.

ASPECTC++. *AspectC++ Homepage*. 2004. Disponível em: <http://www.aspectc.org>.

ASPECTJ. *AspectJ project in Eclipse Projects homepage*. maio 2004. Disponível em: <http://eclipse.org/aspectj/>.

AUGUGLIARO, A. Components designed by contract: A proven, pragmatic, tools-supported process for building real world systems. In: *Open Component Expo 2001*. [s.n.], 2001. Disponível em: <http://ocf.cintec.cuhk.edu.hk/expo2001/>.

BANIASSAD, E.; CLARKE, S. Theme: An approach for aspect-oriented analysis and design. In: *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*. [S.l.: s.n.], 2004.

BASS, L. et al. *Volume I: Market Assessment of Component-Based Software Engineering*. [S.l.], maio 2000. Disponível em: <http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tn007.pdf>.

BAXTER, I. D.; PIDGEON, C. W. Software change through design maintenance. In: *Proceedings of the 13th International Conference on Software Maintenance (ICSM'97)*. [S.l.]: IEEE Computer Society Press, 1997. p. 250–259. ISBN 0-8186-8014-8, 0-8186-8013-X, 0-8186-8015-6.

BAYER, J. Towards engineering product lines using concerns. In: *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*. [s.n.], 2000. Disponível em: <http://www.research.ibm.com/hyperspace/workshops/icse2000/Papers/bayer.pdf>.

BENNETT, K. H.; RAJLICH, V. T. Software maintenance and evolution: a roadmap. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE'2000). Future of Software Engineering Track*. [S.l.]: ACM Press, 2000. p. 73–87. ISBN 1-58113-253-0.

BERGMANN, U.; LEITE, J. From applications domains to executable domains: Achieving reuse with a domain network. In: *Proceedings of the 6th International Conference on Software Reuse (ICSR'2000)*. [S.l.: s.n.], 2000.

BERGMANN, U.; LEITE, J. C. Domain networks in the software development process. *Proceedings of the 7th International Conference on Software Reuse (ICSR'2002), Lecture Notes in Computer Science*, v. 2319, p. 194–209, abr. 2002.

BIANCHI, A. et al. Iterative reengineering of legacy systems. *IEEE Trans. Softw. Eng.*, IEEE Press, v. 29, n. 3, p. 225–241, 2003. ISSN 0098-5589.

BIANCHI, A.; CAIVANO, D.; VISAGGIO, G. Method and process for iterative reengineering of data in a legacy system. In: *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*. [S.l.]: IEEE Computer Society, 2000. p. 86–97. ISBN 0-7695-0881-2.

BIGGERSTAFF, T. J.; MITBANDER, B. G.; WEBSTER, D. E. Program understanding and the concept assignment problem. *Communications of the ACM*, ACM Press, v. 37, n. 5, p. 72–82, 1994. ISSN 0001-0782.

BOOCH, G. *Object-Oriented Analysis and Design with Applications*. Second. Redwood City: Benjamin Cummings, 1994. ISBN 0-8053-5340-2.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *The Unified Modeling Language User Guide*. 1. ed. Reading, Massachusetts, USA: Addison-Wesley, 1999. ISBN 0-201-57168-4.

BOSSONARO, A. A. *Método de Reengenharia de Software Baseada em Componentes (Método RBC)*. Dissertação (Mestrado) — Programa de Pós Graduação em Ciência da Computação, Universidade Federal de São Carlos, 2004.

BOYLE, J. M. Abstract programming and program transformations – an approach to reusing programs. In: BIGGERSTAFF, E. T. J.; J., A. (Ed.). *Software Reusability*. [S.l.]: ACM Press, 1989. I — Concepts and Models, cap. 15, p. 361–413.

BOYLE, J. M.; MURALIDHARAN, M. N. Program reusability through program transformation. *IEEE Transactions on Software Engineering*, v. 10, n. 5, p. 574–588, set. 1984. ISSN 0098-5589. Special Issue on Software Reusability.

BRUEL, J. et al. Using aspects to develop built-in tests for components. In: *The 4th AOSD Modeling With UML Workshop, UML'2003*. [S.l.: s.n.], 2003.

CALDIERA, G.; BASILI, V. R. Identifying and qualifying reusable software components. *IEEE Computer*, v. 24, n. 2, p. 61–71, fev. 1991. ISSN 0018-9162.

CASTOR, F. et al. Jats: A java transformation system. In: *Anais XV Simpósio Brasileiro de Engenharia de Software, Sessão de Ferramentas*. [S.l.: s.n.], 2001.

CHAVEZ, C. *A Model-Driven Approach for Aspect-Oriented Design*. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, abr. 2004.

CHAVEZ, C.; LUCENA, C. Design-level Support for Aspect-Oriented Software Development. In: *Proceedings of the 16th Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'2001). Workshop on Advanced Separation of Concerns in Object-Oriented Systems*. [s.n.], 2001. Disponível em: <http://www.cs.ubc.ca/k̃dvolder/Workshops/OOPSLA2001/submissions/27-chavez.pdf>.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, IEEE Press, v. 20, n. 6, p. 476–493, 1994. ISSN 0098-5589.

CHIKOFSKY, E. J.; CROSS, J. H. Reverse engineering and design recovery: a Taxonomy. *IEEE Software*, v. 1, n. 7, p. 13–17, jan. 1990.

CLARKE, S.; WALKER, R. Towards a standard design language for AOSD. In: *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'2002)*. [S.l.]: ACM Press, 2002. p. 113–119.

CLEMENTE, P.; SÁNCHEZ, F.; PÉREZ, M. Modeling with UML Component-based and Aspect Oriented Programming Systems. In: *Proceedings of the 12th Workshop for PhD Students in Object Oriented Systems. ECOOP Workshops 2002*. [S.l.]: Springer-Verlag, 2002. p. 44–54.

CORDY, J.; CARMICHAEL, I. *The TXL Programming Language Syntax and Informal Semantics*. [S.l.], 1993. v. 7.

CZARNECKI, K.; EISENECKER, U. W.; STEYAERT, P. Beyond Objects: Generative Programming. In: *Proceedings of the 11st European Conference Object-Oriented Programming (ECOOP'97). Workshop on Aspect-Oriented Programming*. [s.n.], 1997. Disponível em: <http://trese.cs.utwente.nl/aop-ecoop97/aop_papers/czarnecki.ps>.

DEURSEN, A. v.; MARIN, M.; MOONEN, L. Aspect Mining and Refactoring. In: UNIVERSITY OF WATERLOO, CANADA. *Proceedings of the First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE03). Held in conjunction with WCRE 2003*. [S.l.], 2003.

DICKINSON, I. J. *Agents Standards*. [S.l.], dez. 09 1997. Disponível em: <http://www.hpl.hp.com/techreports/97/HPL-97-156.html>.

DONNELLY, C.; STALLMAN, R. *Bison. The YACC-compatible Parser Generator*. ago. 2004. Disponível em: <http://dinosaur.compilertools.net/bison/index.html>.

D'SOUZA, D.; WILLS, A. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. First. [S.l.]: Addison-Wesley, 1999. ISBN 0-201-31012-0.

ELRAD, T. et al. Discussing Aspects of AOP. *Communications of ACM*, v. 44, n. 10, p. 33–38, out. 2001a.

ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-Oriented Programming. *Communications of ACM*, v. 44, n. 10, p. 29–32, out. 2001b.

FILMAN, R. E.; FRIEDMAN, D. P. Aspect-oriented programming is quantification and obliviousness. In: *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*. [s.n.], 2000. Disponível em: <http://trese.cs.utwente.nl/Workshops/OOPSLA2000/papers/filman.pdf>.

FONTANETTE, V. et al. Reengenharia de software usando transformações (RST). In: *Proceedings of 2nd Iberoamerican Symposium of Software Engineering and Knowledge Engineering (JIISIC'2002)*. [S.l.: s.n.], 2002a.

FONTANETTE, V. et al. Reprojeto de sistemas legados baseado em componentes de software. In: *Proceedings of infoUYclei 2002 - XXVIII Conferencia Latinoamericano de Informática*. [S.l.: s.n.], 2002b.

FONTANETTE, V. et al. Component-oriented software reengineering using transformations. In: *Proceedings of the ACIS International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications. A Publication of the Internacional Association for Computer and Information Science*. [S.l.: s.n.], 2002c.

FONTANETTE, V. et al. Estratégia de reengenharia de software baseada em componentes distribuídos. In: *Segundo Workshop de Desenvolvimento Baseado em Componentes (WDBC'2002)*. [S.l.: s.n.], 2002d.

FOWLER, M. et al. *Refactoring: improving the design of existing code*. [S.l.]: Addison-Wesley, 1999. (Object Technology Series). ISBN 0-201-48567-2.

FREITAS, F. G.; LEITE, J.; SANT'ANNA, M. Aspectos implementacionais de um gerador de analisadores sintáticos para o suporte a sistemas transformacionais. In: *Simpósio Brasileiro de Linguagens de Programação*. [S.l.: s.n.], 1996. p. 115–127.

FUKUDA, A. P. *Refinamento Automático de Sistemas Orientados a Objetos Distribuídos*. Dissertação (Mestrado) — Programa de Pós Graduação em Ciência da Computação, Universidade Federal de São Carlos, 2000.

GALL, H.; KLÖSCH, R. Capsule oriented reverse engineering for software reuse. *European Software Engineering Conference (ESEC'93), Lecture Notes in Computer Science(LNCSD9)*, v. 717, p. 418–433, 1993. ISSN 0302-9743.

GALL, H.; KLÖSCH, R. Program transformation to enhance the reuse potential of procedural software. In: *Proceeding of the ACM Symposium on Applied Computing (SAC'1994)*. [S.l.]: ACM Press, 1994. p. 99–104. ISBN 0-89791-647-6.

GAMMA, E. et al. *Design Patterns - Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley, 1995. (Addison Wesley Professional Computing Series). ISBN 0-201-63361-2.

GARCIA, V. et al. Em direção a uma abordagem para separação de interesses por meio de mineração de aspectos e refactoring. In: SOLAR, M.; FERNÁNDEZ-BACA, D.; CUADROS-VARGAS, E. (Ed.). *30ma Conferencia Latinoamericana de Informática (CLEI2004)*. [S.l.], 2004a. p. 317–328. ISBN 9972-9876-2-0.

GARCIA, V. C. et al. Uma ferramenta case para o desenvolvimento de software orientado a aspectos. In: *XI Sessão de Ferramentas do XVIII Simpósio Brasileiro de Engenharia de Software (SBES 2004)*. [S.l.: s.n.], 2004b. ISBN 85-7669-004-7.

GARCIA, V. C. et al. Using aspect mining and refactoring to recover knowlegde embedded in object-oriented legacy system. In: *Proceedings of the IEEE International Conference on Information Reuse and Integration (IEEE IRI-2004)*. [S.l.]: IEEE Systems, Man, and Cybernetics Society (SMC), 2004c. p. 30–35. ISBN 0-7803-8819-4.

GARCIA, V. C. et al. Towards an approach for aspect-oriented software reengineering. In: *Proceedings of the 7th International Conference on Enterprise Information Systems (ICEIS-2005).* [S.l.: s.n.], 2005.

GARCIA, V. C. et al. Manipulating Crosscutting Concerns. *4th Latin American Conference on Patterns Languages of Programming (SugarLoafPlop 2004)*, 2004.

GRADECKI, J. D.; LESIECKI, N. *Mastering AspectJ - Aspect-Oriented Programming in Java*. [S.l.]: Wiley Publishing, 2003.

HANENBERG, S.; OBERSCHULTE, C.; UNLAND, R. Refactoring of aspect-oriented software. In: *Net.Object Days 2003*. [s.n.], 2003. Disponível em: <http://www.netobjectdays.org/pdf/03/papers/node/019.pdf>.

HANNEMANN, J.; KICZALES, G. Overcoming the prevalent decomposition in legacy code. In: *Proceedings of the 23rd International Conference on Software Engineering (ICSE'2001). Workshop on Advanced Separation of Concerns in Software Engineering.* [s.n.], 2001. Disponível em: <http://www.research.ibm.com/hyperspace/workshops/icse2001/Papers/hannemann.pdf>.

HARRISON, W.; OSSHER, H. Subject-oriented programming (a critique of pure objects). *ACM SIGPLAN Notices*, v. 28, n. 10, p. 411–428, out. 1993. ISSN 0362-1340.

HERRERO, J. L. et al. Introducing Separation of Aspects at Design Time. In: *Proceedings of the 14th European Conference Object-Oriented Programming (ECOOP'2000). Workshop on Aspects and Dimensions of Concerns*. [S.l.: s.n.], 2000.

HIGHLEY, T.; LACK, M.; MYERS, P. *Aspect Oriented Programming: A Critical Analysis of a New Programming Paradigm*. [S.l.], maio 1999. Disponível em: <http://www.cs.virginia.edu/ techrep/CS-99-29.pdf>.

HILSDALE, E.; KICZALES, G. Aspect-oriented programming with AspectJ. In: . [S.l.: s.n.], 2001. Proceedings of the 16th Object Oriented Programming Systems Languages and Applications (OOPSLA'2001). *Tutorial*. Tampa Bay, FL, USA.

HUGHES, D.; GREENWOOD, P. Aspect testing framework. In: *FMOODS/DAIS 2003 Student Workshop*. [S.l.: s.n.], 2003.

IWAMOTO, M.; ZHAO, J. Refactoring aspect-oriented programs. In: *The 4th AOSD Modeling With UML Workshop, UML'2003*. [S.l.: s.n.], 2003.

JACOBSON, I.; LINDSTROM, F. Reengineering of old systems to an object-oriented architecture. In: *Proceedings of the Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*. [S.l.]: ACM Press, 1991. p. 340–350. ISBN 0-201-55417-8.

JANZEN, D.; De Volder, K. Navigating and querying code without getting lost. In: *Proceedings of the 2nd international conference on Aspect-oriented software development*. [S.l.]: ACM Press, 2003. p. 178–187. ISBN 1-58113-660-9.

JESUS, E.; FUKUDA, A.; PRADO, A. Reengenharia de software para platafomas distribuídas orientadas a objetos. In: *XIII Simpósio Brasileiro de Engenharia de Software (SBES'99)*. [S.l.: s.n.], 1999. p. 289–304.

KELLER, R. K. et al. Pattern-based reverse-engineering of design components. In: *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*. [S.l.]: IEEE Computer Society Press, 1999. p. 226–235. ISBN 1-58113-074-0.

KENDALL, E. A. Reengineering for separation of concerns. In: *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*. [s.n.], 2000. Disponível em: <http://www.research.ibm.com/hyperspace/workshops/icse2000/Papers/kendall.pdf>.

KICZALES, G. et al. Getting started with AspectJ. *Communications of ACM*, v. 44, n. 10, p. 59–65, out. 2001a.

KICZALES, G. et al. An overview of AspectJ. In: KNUDSEN, J. L. (Ed.). *Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP'2001), Lecture Notes in Computer Science (LNCS) 2072*. Berlin: Springer-Verlag, 2001b. p. 327–353.

KICZALES, G. et al. Aspect-Oriented Programming. In: *Proceedings of the 11st European Conference Object-Oriented Programming (ECOOP'97)*. [S.l.]: Springer Verlag, 1997. (LNCS, v. 1241), p. 220–242.

KLEPPE, A. et al. *MDA Explained : The Model Driven Architecture: Practice and Promise*. [S.l.]: Boston, EUA : Addison-Wesley, 2003.

LEE, E. et al. A reengineering process for migrating from an object-oriented legacy system to a component-based system. In: *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC)*. [S.l.]: IEEE Computer Society Press, 2003. p. 336–341.

LEHMAN, M. M.; BELADY, L. A. (Ed.). *Program Evolution: Processes of Software Change*. [S.l.]: Academic Press, 1985. (APIC Studies in Data Processing, v. 27).

LEITE, J. C.; SANT'ANNA, M.; FREITAS, F. G. Draco-PUC: A Technology Assembly for Domain Oriented Software Development. In: *Proceedings of the 3rd International Conference on Software Reuse (ICSR'94)*. [S.l.]: IEEE Computer Society Press, 1994. p. 94–100.

LEITE, J. C.; SANT'ANNA, M.; PRADO, A. F. do. Porting cobol programs using transformational approach. *Journal of Software Maintenance: Research and Practice*, John Wiley & Sons Ltd, v. 9, p. 3–31, out. 1996.

LIEBERHERR, K.; SILVA-LEPE, I.; XAIO, C. Adaptive Object-Oriented Programming Using Graph-Based Customizations. *Communications of the ACM*, ACM Press, v. 37, n. 5, p. 94–101, maio 1994.

LIPPERT, M.; LOPES, C. V. A study on exception detecton and handling using aspect-oriented programming. In: *Proceedings of the 22nd International Conference on Software Engineering*. [S.l.]: ACM Press, 2000. p. 418–427. ISBN 1-58113-206-9.

LUCRÉDIO, D. *Extensão da Ferramenta MVCASE com Serviços Remotos de Armazenamento e Busca de Artefatos de Software*. Dissertação (Mestrado) — Programa de Pós Graduação em Ciência da Computação, Universidade Federal de São Carlos, 2005.

MCILROY, M. D. Mass produced software components. In: *Software Engineering: Report on a conference sponsored by the NATO Science Committee*. [S.l.]: NATO Scientific Affairs Division, 1969. p. 138–155.

MENDONÇA, N. et al. RefaX: A refactoring framework based on XML. In: *Proceedings of the 20th. International Conference on Software Maintenace (ICSM'2004)*. [S.l.]: IEEE Computer Society Press, 2004.

MEYER, B. *Object-Oriented Software Construction*. Second. [S.l.]: Prentice-Hall, Englewood Cliffs, 1997. ISBN 0-13-629155-4.

MONK, S.; HALL, S. *Virtual Mock Objects using AspectJ with JUnit. 2002. XProgramming.com*. jan. 2002. Acessado em: 06/01/2005. Disponível em: <http://xprogramming.com/xpmag/virtualMockObjects.htm>.

MOONEN, L. Exploring software systems. In: *Proceedings of the 19th International Conference on Software Maintenance (ICSM'2003)*. [S.l.]: IEEE Computer Society Press, 2003.

MORAES, J. ao Luis Cardoso de. *Uma Abordagem para Construção e Reutilização de Frameworks de Componentes de Software implementados em Delphi*. Dissertação (Mestrado) — Programa de Pós Graduação em Ciência da Computação, Universidade Federal de São Carlos, 2004.

MÜLLER, H. A. et al. Reverse engineering: a roadmap. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE'2000). Future of Software Engineering Track*. [S.l.]: ACM Press, 2000. p. 47–60. ISBN 1-58113-253-0.

NEIGHBORS, J. *Software Construction Using Components*. Tese (Doutorado) — University of California at Irvine, 1980. Disponível em: <URL: http://www.bayfronttechnologies.com/thesis.htm>.

NEIGHBORS, J. The Draco approach to constructing software from reusable components. In: *IEEE Transactions on Software Engineering. v.se-10, n.5*. [S.l.: s.n.], 1983. p. 567–574.

NEIGHBORS, J. M. Draco: A method for engineering reusable software systems. In: BIGGERSTAFF, T. J.; PERLIS, A. J. (Ed.). *Software Reusability – Concepts and Models*. [S.l.]: ACM Press, 1989. I, p. 295–319.

NEIGHBORS, J. M. Finding reusable software components in large systems. In: *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*. [S.l.]: IEEE Computer Society, 1996. p. 2–10. ISBN 0-8186-7674-4.

NOGUEIRA, A. *Transformação de DataFlex Procedural para Visual DataFlex Orientado a Objetos reusando um Framework*. Dissertação (Mestrado) — Programa de Pós Graduação em Ciência da Computação, Universidade Federal de São Carlos, 2002.

NOVAIS, E. *Reengenharia de Software Orientadas a Componentes Distribuídos*. Dissertação (Mestrado) — Programa de Pós Graduação em Ciência da Computação, Universidade Federal de São Carlos, 2002.

OLSEM, M. R. An incremental approach to software systems re-engineering. *Journal of Software Maintenance*, John Wiley & Sons, Inc., v. 10, n. 3, p. 181–202, 1998. ISSN 1040-550X.

OMG. *XML Metadata Interchange (XMI) Specification*. [S.l.], 2002.

OMONDO. *Omondo. Eclipse UML Tool*. 2004. Disponível em: <http://www.eclipseuml.com/>.

OSSHER, H.; TARR, P. Using subject-oriented programming to overcome common problems in object-oriented software development/evolution. In: *Proceedings of 21st International Conference on Software Engineering (ICSE'99)*. [S.l.]: IEEE Computer Society Press, 1999. p. 687–688.

PAWLAK, R. et al. A UML notation for aspect-oriented software design. In: *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'2002)*. [S.l.]: ACM Press, 2002.

PERES, D. R. et al. TB-REPP - Padrões de Processo para a Engenharia Reversa baseada em Transformações. In: HAMMER, R.; ANDRADE, R. M. C. (Ed.). *The Third Latin American Conference on Pattern Languages of Programming (SugarLoafPLOP 2003)*. [s.n.], 2003. p. 191–215. ISBN 85-87837-08-7. Disponível em: <http://www.cin.ufpe.br/ sugarloafplop/final_articles/12_TB-REPP-Final.pdf>.

PIVETA, E. K.; ZANCANELLA, L. C. Observer pattern using aspect-oriented programming. In: *The Third Latin American Conference on Pattern Languages of Programmng (SugarLoafPLOP 2003)*. [S.l.: s.n.], 2003.

PRADO, A. *Estratégia de Engenharia de Software Orientada a Domínios*. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, jan. 1992.

PRADO, A. F. d. et al. *Reengenharia de Software usando Transformações (RST), RHAE/CNPQ*. [S.l.], mar. 2004.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. Fifth edition. [S.l.]: McGraw-Hill, 2001.

RAMOS, R. A. *Aspecting: Abordagem para Migração de Sistemas OO para Sistemas AO*. Dissertação (Mestrado) — Programa de Pós Graduação em Ciência da Computação, Universidade Federal de São Carlos, 2004.

RATIONAL. *Rational Rose Tool. Rational Software*. 2004. Disponível em: <http://www.rational.com>.

REFINE. *Refine User's Guide. Reasoning Systems Incorporated*. [S.l.], 1992.

RESCUEWARE. *RescueWare. Relativity Technologies*. jan. 2004. Disponível em: <http://www.relativity.com/>.

ROBILLARD, M. P.; MURPHY, G. C. Capturing concern descriptions during program navigation. In: *Workshop on Tools for Aspect-Oriented Software Development (OOPSLA 2002)*. [s.n.], 2002. Disponível em: <http://www.cs.ubc.ca/ murphy/OOPSLA02-Tools-for-AOSD/position-papers/robillard.pdf>.

ROSS, D. T. Structured analysis (SA): A language for communicating ideas. *IEEE Transactions on Software Engineering*, v. 3, n. 1, p. 16–34, jan. 1977. ISSN 0098-5589. Special collection on Requirement Analysis.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. *The Unified Modeling Language Reference Manual*. First. Reading, Massachusetts, USA: Addison-Wesley, 1999. ISBN 0-201-30998-x.

SANT'ANNA, C. et al. On the reuse and maintenance of aspect-oriented software: An evaluation framework. In: *XVII Brasilian Symposium on Software Engineering*. [S.l.: s.n.], 2003.

SNEED, H. M. Planning the reengineering of legacy systems. *IEEE Software*, v. 12, n. 1, p. 24–34, jan. 1995. ISSN 0740-7459.

SNEED, H. M. Object-oriented cobol recycling. In: *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'96)*. [S.l.]: IEEE Computer Society Press, 1996. p. 169–178. ISBN 0-8186-7674-4, 0-8186-7676-0.

SOMMERVILLE, I. *Software Engineering*. Fifth edition. [S.l.]: Addison-Wesley, 1996.

STEIN, D.; HANENBERG, S.; UNLAND, R. An UML-based Aspect-Oriented Design Notation. In: *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'2002)*. [S.l.]: ACM Press, 2002. p. 106–112.

STEVENS, P.; POOLEY, R. Systems reengineering patterns. In: *Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering (FSE-98)*. New York: ACM Press, 1998. (Software Engineering Notes, v. 23, 6), p. 17–23. ISSN 0163-5948.

SUZUKI, J.; YAMAMOTO, Y. Extending UML with aspects: Aspect support in the design phase. In: *Proceedings of 13rd the European Conference Object-Oriented Programming (ECOOP'99). International Workshop on Aspect-Oriented Programming*. [s.n.], 1999. Disponível em: <http://trese.cs.utwente.nl/aop-ecoop99/papers/suzuki.pdf>.

SYSTA, T. The relationships between static and dynamic models in reverse engineering java software. In: *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE'99)*. [S.l.]: IEEE Computer Society Press, 1999.

TARR, P. et al. N degrees of separation: Multi-dimensional separation of concerns. In: *Proceedings of 21st International Conference on Software Engineering (ICSE'99)*. Los Angeles CA, USA: IEEE Computer Society Press, 1999. p. 107–119.

TOGETHER. *Borland Together. Borland Software Corporation*. 2004. Disponível em: <http://www.borland.com/together/>.

UMAR, A. *Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies*. Upper Saddle River, NJ: Prentice-Hall, 1997.

UML. *Unified Modeling Language (UML) - Version 1.5. Object Management Group*. maio 2004. Disponível em: <http://www.uml.org>.

VISAGGIO, G. Ageing of a data intensive legacy system: Symptoms and remedies. *Journal of Soft. Maintenance and Evolution*, v. 13, n. 5, p. 281–308, 2001.

WATERS, R. C. Program translation via abstraction and reimplementation. *IEEE Trans. Softw. Eng.*, IEEE Press, v. 14, n. 8, p. 1207–1228, 1988. ISSN 0098-5589.

WILE, D. *POPART: Producer of Papers and Related Tools System Builders Manual*. [S.l.], 1993.

WILKENING, D. E. et al. A reuse approach to software reengineering. *Journal of Systems and Software*, v. 30, n. 1-2, p. 117–125, 1995. ISSN 0164-1212.

YEH, A. S.; HARRIS, D. R.; REUBENSTEIN, R. Recovering abstract data types and object instances from a conventional procedural language. In: WILLS, L. M.; NEWCOMB, P.; CHIKOFSKY, E. J. (Ed.). *Proceedings: Second Working Conference on Reverse Engineering*. [S.l.]: IEEE Computer Society Press, 1995. p. 227–236. ISBN 0-8186-7111-4.

ZOU, Y.; KONTOGIANNIS, K. Incremental transformation of procedural systems to object oriented platforms. In: *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC)*. [S.l.]: IEEE Computer Society Press, 2003. p. 290–295.

Apêndice A

Durante este trabalho foram publicados diversos artigos, nacionais e internacionais, completos e resumidos. Além disso, também foi obtida uma premiação na Sessão de Ferramentas do Simpósio Brasileiro de Engenharia de Software realizado em Brasília, Outubro de 2004.

Prêmio

Ano: 2004

Indicado para premiação entre as três melhores ferramentas da XI Sessão de Ferramentas do Simpósio Brasileiro de Engenharia de Software (SBES'2004): Uma Ferramenta para o Desenvolvimento de Software Orientado a Aspectos, Sociedade Brasileira de Computação (SBC).

Publicações

Ano: 2005

ALVARO, Alexandre; ALMEIDA, Eduardo Santana de; LUCRÉDIO, Daniel; GARCIA, Vinicius Cardoso; PRADO, Antonio Francisco do; MEIRA, Silvio Romero de Lemos. **ASPECT IPM: Towards an Incremental Process Model Based on AOP for Component-Based Systems**. In the 7th International Conference on Enterprise Information Systems (ICEIS'2005), 24-28 May 2005, Miami, USA. Lecture Notes in Computer Science (LNCS) Springer-Verlag. A ser publicado

GARCIA, Vinicius Cardoso; LUCRÉDIO, Daniel; PRADO, Antonio Francisco do; ALMEIDA, Eduardo Santana de; ALVARO, Alexandre; MEIRA, Silvio Romero de Lemos. **Towards an Approach for Aspect-Oriented Software Reengineering**. In the 7th International Conference on Enterprise Information Systems (ICEIS'2005), 24-28 May 2005, Miami, USA. A ser publicado

Ano: 2004

GARCIA, Vinicius Cardoso; LUCRÉDIO, Daniel; PRADO, Antonio Francisco do; PIVETA, Eduardo Kessler; ZANCANELLA, Luiz Carlos; ALMEIDA, Eduardo Santana de; ALVARO, Alexandre. **Reengenharia de sistemas orientados a objetos através de transformações e minera-**

ção de aspectos. Portuguese/Spanish Tracks In the Third International Information and Telecommunication Technologies Symposium (I2TS'2004), December 6-9, 2004 São Carlos, SP - Brazil.

ALVARO, Alexandre; ALMEIDA, Eduardo Santana de; LUCRÉDIO, Daniel; GARCIA, Vinicius Cardoso. **Aspect IPM: Towards an Incremental Process Model based on AOP for Component-Based Development**. In the First Brazilian Workshop on Aspect-Oriented Software Development (WASP'2004), in conjunction with SBES 2004 Conference, 2004, Brasília, Brazil.

PIVETA, Eduardo Kessler; GARCIA, Vinicius Cardoso; ZANCANELLA, Luiz Carlos; PRADO, Antonio Francisco do. **Termos em português para Desenvolvimento de Software Orientado a Aspectos**. In the First Brazilian Workshop on Aspect-Oriented Software Development (WASP'2004), poster session, in conjunction with SBES 2004 Conference, 2004, Brasília, Brazil.

GARCIA, Vinicius Cardoso; LUCRÉDIO, Daniel; PRADO, Antonio Francisco do; ALVARO, Alexandre; ALMEIDA, Eduardo Santana de. **Using Reengineering and Aspect-based Techniques to Retrieve Knowledge Embedded in Object-Oriented Legacy System**. In the IEEE International Conference on Information Reuse and Integration (IEEE IRI-2004), 2004, Hilton, Las Vegas, Nevada, USA.

ALMEIDA, Eduardo Santana de; ALVARO, Alexandre; LUCRÉDIO, Daniel; GARCIA, Vinicius Cardoso; MEIRA; Silvio Romero de Lemos. **RiSE Project: Towards a Robust Framework for Software Reuse**. In the IEEE International Conference on Information Reuse and Integration (IEEE IRI-2004), 2004, Hilton, Las Vegas, Nevada, USA.

GARCIA, Vinicius Cardoso; LUCRÉDIO, Daniel; ALVARO, Alexandre; ALMEIDA, Eduardo Santana de; PRADO, Antonio Francisco do. **Towards an effective approach for Reverse Engineering**. In the 11th IEEE Working Conference on Reverse Engineering (WCRE 2004 Conference), 2004, Delft University of Technology, Netherlands.

GARCIA, Vinicius Cardoso; PIVETA, Eduardo Kessler; LUCRÉDIO, Daniel; ALVARO, Alexandre; ALMEIDA, Eduardo Santana de; PRADO, Antonio Francisco do; ZANCANELLA, Luiz Carlos. **Em direção a uma abordagem para separação de interesses por meio de Mineração de Aspectos e Refactoring**. In XXX LATIN AMERICAN CENTER OF STUDIES IN COMPUTER SCIENCE (CLEI 2004 Conference), 2004, Arequipa, Peru.

GARCIA, Vinicius Cardoso; PRADO, Antonio Francisco do. **Phoenix: Uma Abordagem para a Reengenharia de Software Orientada a Aspectos**. In IX Workshop de Teses e Dissertações em Engenharia de Software - XVIII Simpósio Brasileiro de Engenharia de Software. (SBES 2004 Conference), 2004, Brasília, Brazil.

GARCIA, Vinicius Cardoso; LUCRÉDIO, Daniel; FROTA, Luíza; ALVARO, Alexandre; ALMEIDA, Eduardo Santana de; PRADO, Antonio Francisco do. **Uma ferramenta CASE para o Desenvolvimento de Software Orientado a Aspectos**. In XI Sessão de Ferramentas do XVIII Simpósio Brasileiro de Engenharia de Software. 2004. (SBES 2004 Conference), 2004, Brasília, Brazil.

GARCIA, Vinicius Cardoso; PIVETA, Eduardo Kessler; LUCRÉDIO, Daniel; ALVARO, Alexandre; ALMEIDA, Eduardo Santana de; PRADO, Antonio Francisco do; ZANCANELLA, Luiz Carlos. **Manipulating Crosscutting Concerns**. In The Fourth Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP 2004 Conference), 2004, Porto das Dunas-CE, Brazil.

LUCRÉDIO, Daniel; ALMEIDA, Eduardo Santana de; GARCIA, Vinicius Cardoso; ALVARO, Alexandre; PIVETA, Eduardo Kessler. **Student's PLoP Guide: A Pattern Family to Guide Computer Science Students during PLoP Conferences**. In The Fourth Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP 2004 Conference), 2004, Porto das Dunas-CE, Brazil.

MORAES, João Luis Cardoso de; BOSSONARO, Adriano Aleixo; FONTANETTE, Valdirene; LUCRÉDIO, Daniel; GARCIA, Vinicius Cardoso; PRADO, Antonio Francisco do. **An Approach for Construction and Reuse of Software Components Frameworks implemented in Delphi**. In Proceedings of The Fourth International School and Symposium on Advanced Distributed Systems (ISSADS'2004), 2004, March 8-10 2004 in Guadalajara Jalisco, Mexico.

Ano: 2003

ALVARO, Alexandre; LUCRÉDIO, Daniel; GARCIA, Vinicius Cardoso; ALMEIDA, Eduardo Santana de; PRADO, Antonio Francisco do; TREVELIN, Luis Carlos. **Orion-RE: A Component-Based Software Reengineering Environment**. In Proceedings of The 10th Working Conference on Reverse Engineering (WCRE'2003), 2003, Victoria, British Columbia, Canada.

BOSSONARO, Adriano Aleixo; MORAES, João Luis Cardoso de ; FONTANETTE, Valdirene; GARCIA, Vinicius Cardoso; PRADO, Antonio Francisco do. **Implementações de Frameworks de Componentes, dirigidas por Modelos do Método Catalysis**. In Proceedings of The Fourth Congress of Logic Applied to Technology (LAPTEC'2003), 2003, Marília-SP, Brazil.

PERES, Darley Rosa; ALVARO, Alexandre; FONTANETTE, Valdirene; GARCIA, Vinicius Cardoso; PRADO, Antonio Francisco do. **TB-REPP - Padrões de Processo para a Engenharia Reversa baseada em Transformações**. In The Third Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP 2003 Conference), 2003, Porto de Galinhas-PE, Brazil.