Desenvolvimento de Software Orientado a Temas: Um Estudo de Caso

Antonielly Garcia Rodrigues

Desenvolvimento de Software Orientado a Temas: Um Estudo de Caso

Antonielly Garcia Rodrigues

Orientador: ***Dr. Paulo Cesar Masiero***

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional.

USP - São Carlos
Março/2006

Agradecimentos

Agradeço às seguintes pessoas e instituições por mais esta fase concluída com sucesso em minha vida:

Minha mãe, Bernadete, meus tios Renato e Regina, e meus avôs Antônio e Helena, por fazerem a vaquinha para pagar minhas despesas na época do início do Mestrado, um período de vacas magras, enquanto eu não conseguia bolsa.

Professores da UFG Eduardo Simões, Fábio Moreira, Fábio Nogueira, Márcio e Juliano, pela disposição em redigir as cartas de recomendação que foram decisivas para que eu fosse selecionado ao programa de Mestrado.

Valter Camargo, por ter cedido sua monografia de qualificação de Doutorado para que eu pudesse utilizar como base para o plano de pesquisa, e pelas várias discussões sobre pesquisa.

Marcelo Eler e Darley, pelas enormes contribuições nos trabalhos de disciplinas.

Otávio Lemos, pelas discussões sobre pesquisa e sobre filosofia, pela amizade e também pelas caronas para casa depois de dias inteiros no laboratório.

Júlio Cezar, por ter me emprestado o computador dele para redigir a maior parte desta dissertação, já que no laboratório eu não conseguia um bom rendimento.

Harold Ossher, por haver gentilmente me enviado uma versão extra-oficial do compilador de *Hyper/J* que não tinha um defeito encontrado por mim na versão oficial.

Pessoal que frequentou o WASP em 2004 e 2005, em especial à Christina Chavez, pelas discussões, mesmo considerando que, sob a empolgação, eu sempre acabava falando de mais e ouvindo de menos.

Meu orientador Paulo Cesar Masiero, que, por sua filosofia gerencial de cobrar resultados semanalmente, fez com que a pesquisa progredisse, já que isso me tirava do estado *default* de comodismo e procrastinação em que eu, como um típico ser humano, era tentado a permanecer.

Rosely Sanches, pelas dicas a respeito de como funciona o meio científico e pelo incentivo em minha tentativa de publicar um artigo sobre um assunto diferente daquele em que meu Mestrado foi focado.

Turma da *PgCompUsp04* (colegas que entraram na mesma época que eu na Pós e colegas de outros anos e de outros cursos que foram "adotados"), por ter sido uma galera muito animada que me proporcionou a maioria dos momentos bons ao longo deste Mestrado, especialmente no início do curso, em que eu não conhecia quase ninguém na cidade, e nas saídas para tomar garapas, assistir filmes, frequentar rodízios de pizza, comer sanduíches, participar de churrascadas, ir à festas (no CAASO, no Café Cancun e no Moinho Santa Maria), ir a bares e espetinhos, assistir a espetáculos (Ira e Patofu), disputar corrida, e tocar música nas Jam Sessions.

Jovens que moraram comigo no Pensionato Sakura, pelas conversas engraçadas na sala e no corredor, pelos almoços "públicos" (coletivos) e pela companhia nas idas ao cineclube gratuito da USP.

Shigemi Shigenaga, pela boa convivência e pelas gentilezas, como dar carona para a Rodoviária, enquanto eu morava em seu pensionato.

Galerinha da República Enxofre, que me "adotou" quando meu contrato com o Pensionato Sakura expirou.

Amigos de Goiás, que mesmo geograficamente longe me davam o carinho e o apoio que eu precisava para superar as carências e dificuldades em meu difícil processo de adaptação ao novo estilo de vida, em outra cidade e sem a família por perto.

CNPq e FAPESP, por terem dado suporte financeiro à minha subsistência durante quase todo o período de Mestrado, com base na crença de que meu trabalho contribuiria com o avanço da Ciência.

Pelas demais pessoas que compartilharam momentos comigo ao longo do Mestrado, cujos nomes não irei citar aqui em razão do receio de omissão, por terem proporcionado situações que aproveitei para crescer como ser humano.

Le savant n'est pas l'homme qui fournit les vraies réponses; c'est celui qui pose les vraies questions.
(Claude Lévi-Strauss)

Resumo

O Paradigma Orientado a Objetos tem sido atualmente a abordagem dominante de desenvolvimento de software. Contudo, ela sofre da Tirania da Decomposição Dominante, pois não permite uma modularização adequada da implementação relativa a interesses estruturais. Como consequência, a implementação relativa a cada interesse estrutural fica espalhada pelos módulos do programa e entrelaçada com a implementação relativa a outros interesses estruturais. Outras abordagens de desenvolvimento de software, como o Desenvolvimento de Software Orientado a Aspectos com *AspectJ* e a Separação Multidimensional de Interesses em Hiperespaços com *Hyper/J* e CME, atingem sucesso moderado em oferecer mecanismos que permitem superar as deficiências do Paradigma Orientado a Objetos. No entanto, tais abordagens também possuem deficiências e omissões que devem ser reparadas para que elas possam se tornar utilizáveis em contextos típicos de desenvolvimento de software complexo. Este trabalho especifica uma nova abordagem, denominada Desenvolvimento de Software Orientado a Temas (DSOT), que tem como objetivo superar algumas deficiências das abordagens anteriores por meio de mecanismos que permitem a manipulação da implementação de cada interesse estrutural de forma separada e a manipulação da implementação de cada tipo de dado de forma separada. Além disso, DSOT possui operadores que são ortogonais, isto é, podem ser utilizados de forma combinada ou separada, para efetuar a composição de módulos do programa. Mostra-se o modelo conceitual do DSOT e descreve-se um estudo de caso que consiste no desenvolvimento de um programa para demonstrar mais concretamente como o DSOT funciona na prática. Não se demonstra a superioridade do DSOT para o caso geral, mas os resultados alcançados evidenciam que o DSOT é uma abordagem promissora que merece ser investigada mais aprofundadamente em pesquisas futuras.

Abstract

The Object-Oriented Paradigm has currently been the dominant approach for developing software. However, it suffers from the Tyranny of the Dominant Decomposition, as it does not support a suitable modularization to the implementation relative to structural concerns. As a consequence, the implementation relative to each structural concern is scattered throughout the program modules and tangled with the implementation relative to other structural concerns. Some software development approaches, such as Aspect-Oriented Software Development with Aspect and Multidimensional Separation of Concerns in Hyperspaces with Hyper/J and CME, achieve moderate success in offering mechanisms that make it possible to overcome the deficiencies of the Object-Oriented Paradigm. However, such approaches also possess deficiencies and ommissions that must be corrected in order for them to get usable in typical complex software development contexts. This work specifies a new approach, named Theme-Oriented Software Development (TOSD), which aims at overcoming some deficiencies from previous approaches through mechanisms that support the handling of implementation for every structural concern separately and the handling of implementation for every data type separately. Moreover, TOSD contains operators which are orthogonal, that is, they can be used separately or as a combination, in order to perform composition of the program modules. We show the conceptual model of TOSD and describe a case study which consists in the development of a program to demonstrate more concretely how TOSD works in practice. We do not demonstrate the superiority of TOSD for the general case, but the results we have obtained suggest that TOSD is a promissing approach which deserves a deeper investigation in future research.

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO

1

Introdução

1.1 Contextualização

No campo de Engenharia de Software, várias abordagens imperativas têm sido criadas ao longo dos anos como resposta a dificuldades em desenvolver e evoluir software. Em geral, uma abordagem surge a partir do reconhecimento de deficiências inerentes à abordagem anterior para evoluir software e da criação de uma nova maneira de abstrair partes do programa. Essas observações são feitas por meio da análise do processo de desenvolvimento e evolução de programas, e também por meio da comparação do *design* entre versões diferentes do mesmo programa.

A natureza dos requisitos para a evolução de um sistema sugere que cada requisito envolve um conjunto de elementos de informação relacionados de alguma forma. Portanto, a satisfação do requisito provavelmente causará impacto nos pontos da estrutura do programa que de alguma forma manipulam alguns daqueles elementos de informação. A esse conjunto de elementos de informação relacionados dá-se o nome de interesse (Dijkstra, 1976). Se a implementação relativa a um determinado interesse puder ser separadamente modularizada no programa, então a mudança que envolve tal interesse poderá ser mais restringida.

A comparação entre versões de um mesmo programa permite encontrar pontos do *design* onde ocorreram mudanças invasivas, isto é, alterações necessárias em módulos existentes na versão anterior. É importante encontrar mudanças invasivas porque elas são sintomas de uma separação de interesses inadequada no *design* do programa (Tarr et al., 1999).

Uma separação de interesses inadequada pode ocorrer por causa da falta de atenção, motivação ou conhecimento do desenvolvedor para criar um *design* bom o suficiente. Esse tipo de problema

pode ser resolvido com a aplicação de uma sequência de refatorações sobre o sistema até que o *design* se torne satisfatório. Pode também ocorrer um problema desse tipo se a abordagem de desenvolvimento não fornecer mecanismos que permitam a separação adequada de determinados interesses. Nesse caso, não é possível criar uma solução satisfatória por meio do uso exclusivo dos mecanismos de modularização daquela abordagem. É essa segunda maneira que motiva o surgimento de uma nova abordagem.

Quando se procura definir o modelo conceitual de uma nova abordagem, é necessário estudar as propostas mais notáveis de mecanismos para estender a abordagem antiga, tomando nota de vantagens, deficiências e omissões, que podem ser aproveitadas como lição para se criar uma proposta com base teórica melhor estabelecida. Então devem ser desenvolvidos alguns programas para experimentar e refinar os mecanismos que parecem ser necessários na nova abordagem, e também para adquirir evidências concretas de que o uso de tais mecanismos é de fato vantajoso. Após alguns ciclos de experimentação e refinamento, uma nova abordagem começa a surgir. Esse processo funciona ainda melhor se a base teórica em desenvolvimento é apresentada e colocada em discussão com outras pessoas que possuam conhecimento de assuntos relacionados.

1.2 Motivações

Atualmente, o Paradigma de Orientação a Objetos (POO) tem sido a abordagem dominante de desenvolvimento e evolução de software. Das abordagens imperativas que estão efetivamente maduras para uso prático, o POO é a mais avançada. Seu uso intensivo e sua grande base de usuários faz com que suas deficiências se tornem naturalmente melhor compreendidas com o passar do tempo. A melhor compreensão dessas deficiências tem começado a abrir espaço para o surgimento de novas abordagens, como o Desenvolvimento de Software Orientado a Aspectos com *AspectJ* (Kiczales et al., 1997; The AspectJ Team, 2006) e a Separação Multidimensional de Interesses em Hiperespaços com *Hyper/J* e *CME* (Tarr et al., 1999; Ossher e Tarr, 2001; Clarke e Baniassad, 2005; Tarr e Ossher, 2000; Ossher e Tarr, 1999).

Uma das deficiências do Paradigma de Orientação a Objetos é o espalhamento sobre várias rotinas e classes da implementação relativa a determinados interesses, acompanhada do entrelaçamento da implementação relativa a um determinado interesse com a implementação relativa a outro interesse (Kiczales et al., 1997). Um interesse cuja implementação não é modularizada adequadamente no Paradigma de Orientação a Objetos é denominado interesse estrutural.

O Paradigma de Orientação a Objetos possui mecanismos adequados somente para modularizar interesses de dois tipos: operações e tipos de dados. A implementação de interesses estruturais não é bem modularizada no Paradigma de Orientação a Objetos porque ela não fornece um mecanismo complementar para visualizar e manipular separadamente a implementação de interesses estruturais. Esse fenômeno é denominado Tirania da Decomposição Dominante (Tarr et al., 1999).

As abordagens que surgiram para superar as deficiências do Paradigma de Orientação a Objetos conseguem atingir algum progresso. Todavia, elas não estão suficientemente maduras com relação aos mecanismos fornecidos para auxiliar a modularização dos programas. *AspectJ* fornece um mecanismo para manipular separadamente a implementação de interesses estruturais. Entretanto, perde-se a capacidade original do Paradigma de Orientação a Objetos de visualizar e manipular a implementação combinada dos vários interesses estruturais em classes e rotinas. O problema de *Hyper/J* e CME é diferente. Ambos permitem a manipulação da implementação relativa a cada interesse estrutural separadamente ou de forma entrelaçada à implementação de outros interesses estruturais, conforme o desejo do desenvolvedor. No entanto, os operadores fornecidos para auxiliar a compor os módulos dos programas não são ortogonais, isto é, não podem ser combinados ou utilizados separadamente uns dos outros.

Isso significa que deve ser criada uma nova abordagem que permita a manipulação separada ou entrelaçada da implementação de interesses estruturais e também forneça operadores ortogonais para a composição dos módulos dos programas. A relevância deste trabalho é compreendida quando se percebe a imensa quantidade de sistemas de software que atualmente são desenvolvidos com o uso do Paradigma Orientado a Objetos. A atualidade do tema é percebida quando se considera o ritmo ativo de pesquisas na área de Desenvolvimento de Software Orientado a Aspectos feitas pela comunidade de pesquisadores em Engenharia de Software.

1.3 Objetivos

Os principais objetivos deste trabalho são:

- Especificar o modelo conceitual de uma abordagem que permita superar algumas deficiências inerentes à estruturação de programas nas abordagens Paradigma Orientado a Objetos, Desenvolvimento de Software Orientado a Aspectos com *AspectJ* e Separação Multidimensional de Interesses em Hiperespações com *Hyper/J* e *CME*;
- Descrever um estudo de caso que consiste no desenvolvimento de um programa não-trivial na abordagem especificada.

Para atingir esses objetivos, esta dissertação é estruturada conforme mencionado na próxima seção.

1.4 Organização

O restante desta dissertação está organizado da seguinte maneira. No Capítulo 2, são mostradas as idéias que originaram algumas das abordagens imperativas de modularização de software mais

importantes. Em seguida, no Capítulo 3 são discutidas as características principais do modelo conceitual do Desenvolvimento de Software Orientado a Temas, incluindo a semântica de composição e de redefinição. Então, descreve-se no Capítulo 4 um estudo de caso com o intuito de demonstrar de maneira mais concreta como funcionam os mecanismos do Desenvolvimento de Software Orientado a Temas. No Capítulo 5, avalia-se as capacidades da abordagem de DSOT em comparação com outras abordagens. Finalmente, são expostas no Capítulo 6 as conclusões desta pesquisa e uma lista de trabalhos futuros.

CAPÍTULO

2

Evolução Conceitual das Abordagens Imperativas de Modularização de Software

2.1 Considerações Iniciais

O objetivo deste capítulo é mostrar o contexto geral das abordagens imperativas de desenvolvimento de software estabelecidas antes do Desenvolvimento de Software Orientado a Temas, a fim de situar a abordagem tratada nesta dissertação no contexto maior de pesquisa na área de Engenharia de Software. Não se pretende aqui descrever com rigor histórico a demonstração do desenvolvimento gradual das idéias, mas apenas uma explicação bastante simplificada de como a história teria sido se as descobertas tivessem realmente ocorrido de forma linear. Interessa-se aqui apenas pelas contribuições, em um nível conceitual, de cada abordagem.

As abordagens imperativas obedecem ao modelo computacional de Von Neumann, contendo assim os conceitos de: variáveis e objetos, que representam células de memória; atribuição a e leitura de variáveis; algoritmos do programa como uma sequência de instruções fornecidas a um computador abstrato. As abordagens imperativas se distinguem de abordagens declarativas, como a da linguagem Prolog, e funcionais, que seguem a linha de LISP, que não são tratadas nesta dissertação.

Este capítulo está organizado como se segue. Discutem-se na Seção 2.2 algumas observações sobre evoluibilidade de software, o conceito que justifica a distinção entre as várias abordagens

de desenvolvimento. Em seguida, aparece na Seção 2.3 o conceito de separação de interesses, fundamental para a boa estruturação de programas em quaisquer abordagens. Como o Paradigma Orientado a Objetos tem sido a abordagem imperativa dominante atualmente em termos de uso prático, optou-se por dividir o restante deste capítulo em duas grandes seções. Na Seção 2.4 são comentadas as abordagens imperativas clássicas, que vão do Paradigma Pré-Procedimental até o Paradigma Orientado a Objetos. Na Seção 2.5 são apresentadas algumas abordagens que surgiram depois do Paradigma Orientado a Objetos e que possuem uma relação histórica com o Desenvolvimento de Software Orientado a Temas.

2.2 Evoluibilidade do Software

É um fato que todo software bem sucedido está sujeito a pressões para que evolua (Brooks, 1987). Um software bem desenhado é utilizado para fins muitas vezes não antecipados pelos desenvolvedores, que devem então efetuar mudanças no software para que ele comporte novas características e funcionalidades requeridas por seus usuários.

Um sistema de software é inerentemente complexo, e sua evolução exige bastante conhecimento sobre a sua estruturação modular e grande cautela para que não a harmonia do *design* não seja destruída por modificações mal pensadas e para que defeitos não sejam gerados pela imprudência ou falha humana do desenvolvedor. Como a evolução do software é regra e não exceção para a maioria dos produtos de software criados, deseja-se que as mudanças causem o mínimo de impacto possível sobre o *design* existente.

A evoluibilidade é um conceito multidimensional formado por alguns fatores. Se os módulos existentes forem compreensíveis, torna-se menos provável efetuar mudanças que tornam o software defeituoso de maneira não-intencional. Se houver módulos disponíveis para reuso que implementam as mudanças necessárias, tais módulos podem ser acoplados no *design* existente, simplificando o processo de modificação. Se as mudanças necessárias afetarem somente uma pequena parte do *design* existente, as modificações tornam-se mais fáceis e menos propensas a erros. Assim, os fatores de compreensibilidade, reusabilidade e modificabilidade são os principais componentes da evoluibilidade.

Para um sistema possuir melhor evoluibilidade, é importante que as mudanças ao longo do tempo sejam, em sua maior parte, aditivas, e não invasivas (Tarr et al., 1999). A aditividade contribui para limitar o escopo das mudanças necessárias: menos módulos serão afetados e portanto será reduzida a probabilidade do desenvolvedor cometer erros de *design*. O ideal é que os módulos do programa tenham a propriedade de adaptabilidade não-invasiva, que é a possibilidade de adaptar um módulo sob demanda sem modificá-lo invasivamente, mesmo se a adaptação não tiver sido planejada na época em que foi criado o *design* original do módulo (Czarnecki e Eisenecker, 2000). Assim, seria possível implementar os vários tipos de mudanças requeridas ao longo do ciclo de vida do software sem afetar a implementação dos módulos que já existem. A adaptabilidade não-

invasiva permite que se evolua um sistema de software por meio de mudanças aditivas, evitando assim ao máximo a necessidade de refatorações não-essenciais.

As técnicas de refatoração podem ser encaradas como maneiras disciplinadas de efetuar mudanças infelizmente invasivas. É vantajoso reduzir a necessidade de refatorações, já que o ato de modificar a estrutura de um software já existente possui riscos, como a possibilidade de cometer não-intencionalmente erros durante a refatoração, e ônus, como a necessidade de testes de regressão e da criação de novos testes. Contudo, a adaptabilidade não-invasiva minimiza, mas não elimina, necessidades ocasionais de refatoração em uma futura evolução do sistema. Por exemplo, quando o desenvolvedor acabou de identificar um interesse cuja implementação está espalhada ao longo do sistema, e pretende localizar a implementação daquele interesse em um único módulo, a refatoração é uma técnica valiosa. Outra importante aplicação de refatoração seria em um cenário no qual alguns módulos teriam ocasionalmente de ser remodelados a fim de melhor se adequarem ao conceito geral do sistema, por causa de erros de *design*, necessidade de pequenos retoques, generalizações tardias para aumentar a reusabilidade dos módulos, ou outras razões. O que se deseja não é reduzir a quantidade de refatorações para melhorar o *design* do sistema, já que dificilmente o desenvolvedor conseguiria criar um *design* totalmente adequado logo na primeira tentativa. Mas deseja-se reduzir a necessidade de mudanças extremas no *design* do sistema, e em consequência a quantidade de refatorações necessárias para implementar tais mudanças extremas.

Todavia, infelizmente mudanças aditivas são a exceção, e não a regra. É importante determinar porque isso ocorre, e o que fazer para reverter esse quadro. Uma das principais causas do grande impacto das mudanças é a falta de correspondência entre o escopo das mudanças necessárias e a estrutura modular do programa (Tarr et al., 1999). A atividade de desenhar a estrutura do programa é uma responsabilidade do desenvolvedor. Uma maneira de possibilitar a ocorrência de mudanças aditivas futuras no programa é que o desenvolvedor antecipe tais mudanças e as habilite pelo uso adequado de padrões de *design*, de boas arquiteturas, ou de um *design* mais geral, na medida do que é possibilitado pelos mecanismos fornecidos pela abordagem de modularização utilizada ao longo do desenvolvimento do programa (Ossher e Tarr, 2001). Isso significa que o desenvolvedor teria de estabelecer vários ganchos de extensão no *design* do programa. Entretanto, nem mesmo o desenvolvedor mais competente poderia efetuar tais atividades com sucesso, pelos fatores a seguir.

Em primeiro lugar, grande parte da evolução, reuso e integração de software não podem ser antecipados. Isso não ocorre necessariamente por causa de um *design* ruim, mas porque as informações e necessidades mudam tão rapidamente que se torna praticamente impossível prever os caminhos da evolução e do uso do software com acurácia. Além disso, o tempo em um projeto frequentemente é um recurso escasso, o que impede o desenvolvedor de investir tempo suficiente para antecipar algumas mudanças, já que o tempo está alocado para outras atividades de maior prioridade. Portanto, este tipo de cenário de evolução, no qual uma mudança não esperada é necessária, é comum e importante (Ossher e Tarr, 2001).

Outro fator que impede o uso de tal estratégia é que, mesmo para as mudanças que podem ser corretamente antecipadas, seria indesejável fornecer todo gancho de extensão possível: o custo

de fazê-lo seria proibitivo, tanto em termos de complexidade adicional quanto em desempenho de execução reduzido. Um *design* mais geral e mais flexível tende a ser mais complicado de usar do que um *design* mais específico para o propósito em questão. A inserção de ganchos de extensão incorre em alguma queda de desempenho até mesmo quando eles não são usados. Além do mais, ocorre o paradoxo interessante de que um *design* flexível para determinado tipo de mudança pode ser mais complicado de alterar do que um *design* simples, em especial nos cenários em que outro tipo de mudança necessária, que não foi antecipada, deve ser feito no programa (Ossher e Tarr, 2001). Assim, conclui-se que criar um *design* altamente flexível não é a melhor saída para melhorar a evoluibilidade do programa.

Se for possível dividir o programa em módulos de forma que o escopo das mudanças se alinhe com a estrutura modular do programa, a evoluibilidade será bastante melhorada. Mas para isso a abordagem de desenvolvimento deve em primeiro lugar fornecer os tipos de módulos necessários e uma semântica adequada para integrá-los. Assim, o que se deve fazer é melhorar os mecanismos de modularização da abordagem de desenvolvimento para que ela facilite o desenvolvimento e a evolução de programas.

Para descobrir quais mecanismos de modularização são necessários, é importante analisar porque algumas mudanças são inerentemente invasivas quando se usa um determinado mecanismo de modularização, e deve-se verificar se é possível utilizar outro mecanismo, já existente ou ainda a ser criado, para reduzir a necessidade de mudanças invasivas.

A natureza dos requisitos para a evolução de um sistema é a de que cada requisito envolve um conjunto de elementos de informação relacionados de alguma forma. Assim, a satisfação do requisito tenderá a causar impacto nos pontos da estrutura do programa que envolvem tais elementos. Se a implementação que processa esse conjunto de elementos relacionados puder ser separadamente modularizada no programa, então a mudança que envolve tais elementos de informação poderá ser aditiva, ou invasiva em somente um módulo, em vez de afetar várias porções distintas do *design*. Este conjunto de elementos de informação relacionados é denominado *interesse*. Então, o objetivo do desenvolvedor é efetuar a modularização do programa de maneira a obter uma *separação de interesses* adequada (Dijkstra, 1976).

2.3 Separação de Interesses

Um problema complexo é resolvido por meio de um *processo* de solução de problemas, cujo resultado esperado é uma solução. Argumenta-se que o poder cognitivo humano é suficiente para resolver muitas categorias de problemas complexos, desde que se apliquem estratégias cognitivas adequadas desde o início da busca por uma solução satisfatória.

Contudo, uma estratégia cognitiva arbitrária não possibilita em geral atingir sucesso na solução de problemas. Deve-se considerar as várias limitações da poderosa mente humana que são cientificamente reconhecidas há bastante tempo (Miller, 1956). Por exemplo, as pessoas possuem difi-

culdade de compreender e de processar muita informação com interrelacionamentos complexos de uma só vez. Reconhecendo tais limitações, Dijkstra (1976) sugeriu, originalmente com o propósito de facilitar o desenvolvimento de software, uma estratégia geral que pode ser utilizada por humanos para lidar com a complexidade arbitrária. A idéia é que a compreensão torna-se mais fácil quando se foca a cada momento a atenção em apenas um aspecto auto-contido, ou interesse, tendo-se ciência de que se está ignorando temporariamente os outros aspectos do problema. Esta estratégia geral é conhecida como *Separação de Interesses*.

A estratégia cognitiva de Separação de Interesses sugere que, a fim de se resolverem problemas complexos, é necessário:

- identificar ou desentrelaçar os aspectos importantes de um problema de forma que eles se sobreponham o mínimo possível, e
- analisar cada aspecto separadamente, abstraindo a existência de outros aspectos, em vez de manipular diretamente a mistura confusa de elementos de informação.

Assim, ao adotar uma estratégia no qual interesses são plenamente separados, não é necessário lidar com toda a complexidade de uma só vez, embora o conjunto represente a complexidade total do problema. A melhoria da compreensibilidade advém do fato de que a separação de interesses converte a complexidade caótica em complexidade intelectualmente gerenciável.

Em certo sentido, Separação de Interesses é uma maneira diferente de compreender o velho conceito de abstração. A diferença é que a Separação de Interesses enfatiza mais a identificação dos interesses e o isolamento das propriedades referentes a cada um deles, ao passo que a abstração destaca a necessidade de se ignorar outros interesses de um problema enquanto se analisa um interesse separadamente dos outros. Na prática, estes dois processos se intercalam e se confundem.

Um desafio importante que aparece quando se aplica a estratégia mencionada é que *interesses que se sobrepõem* são inerentes a muitos problemas complexos. Mesmo se uma separação completa de interesses não for possível, já que interesses podem se sobrepor uns aos outros, deve-se tentar separar os interesses o máximo possível, já que é o melhor que se pode fazer para facilitar a compreensão.

Em Engenharia de Software, a separação de interesses para auxiliar o desenvolvimento inicial é importante por permitir aos desenvolvedores gerenciar a complexidade do software. Contudo, a separação de interesses para promover evolução, integração e reuso é ainda mais crítica, já que a maioria do esforço é despendido nessas atividades (Ossher e Tarr, 2001).

Assim, um sistema de software bem delineado deve ser, dentre outras coisas, estruturado em módulos compreensíveis de forma que cada módulo:

- seja capaz, em tempo de execução, de interagir ordenadamente com outros módulos para gerar o comportamento requerido,

- encapsule *toda* a implementação de exatamente *um* interesse.

Toda abordagem de modularização fornece infra-estrutura para habilitar, em algum grau, a separação de interesses. Tal infra-estrutura consiste em geral de módulos (mecanismos de decomposição) para separar a implementação de cada interesse considerado da implementação dos demais interesses, e de mecanismos de composição para integrar adequadamente os módulos a fim de formar componentes maiores ou sistemas completos. O que distingue cada abordagem são os tipos de interesses que cada uma delas permite separar adequadamente.

A existência de uma abordagem de modularização adequada, contudo, é uma condição necessária mas não suficiente para atingir uma separação conveniente de interesses. O desenvolvedor também possui um papel-chave para esse fim. O *design* subjacente da aplicação é mais importante do que os mecanismos fornecidos por abordagens de modularização, independentemente de sua sofisticação. Isto é, uma boa separação de interesses é garantida, em última instância, pela arquitetura do sistema. Tecnologias são úteis nesse contexto, por promoverem boas práticas de desenvolvimento, por exemplo tornando a implementação de arquiteturas adequadas possível, e por fornecerem os construtos necessários para separar adequadamente os interesses. (Pace e Campo, 2001).

A próxima seção comenta sobre as abordagens imperativas mais tradicionais de modularização de software.

2.4 Abordagens Imperativas Clássicas

Cada abordagem imperativa que surge tem como objetivo fornecer mecanismos para modularizar a implementação relativa a um tipo de interesse que a abordagem anterior não era capaz de tratar adequadamente. Para mostrar que tipos de interesses foram tratados ao longo do tempo, será mostrado como pode ser encarada a evolução das abordagens imperativas clássicas.

Algumas linguagens de programação possuem vários tipos de módulos e artifícios, maneiras distintas de considerar tipagem, semânticas diferentes para passagem de parâmetros e instanciação de objetos, e outras idiossincrasias específicas. Além disso, várias linguagens possuem, muitas vezes para atingir maior eficiência em detrimento de outros fatores, detalhes que as vinculam excessivamente à tecnologia dos computadores físicos (por exemplo, a ausência de um mecanismo de coleta de lixo e a existência de mecanismos de desalocação manual de memória). Tais detalhes não serão tratados nesta discussão. A discussão presente nessa seção sobre abordagens não é necessariamente ligada a nenhuma linguagem de programação específica. Sendo assim, os conceitos e estratégias aqui apresentados podem ser diferentes dos mecanismos que as várias linguagens de programação existentes fornecem.

Ao longo da evolução dos paradigmas e das linguagens de programação, sempre houve uma tensão em saber quais elementos devem ser estaticamente tratados e quais elementos devem ser

considerados em um contexto dinâmico. Muitas vezes a determinação depende do propósito da linguagem: a ênfase maior pode ser dada em confiabilidade, em flexibilidade, em eficiência ou em outros fatores. Na discussão presente neste capítulo, o seguinte cenário será considerado:

- A alocação de memória para objetos ocorre somente de forma dinâmica no *heap*; não há alocação dinâmica por pilha ou alocação estática.
- Há um mecanismo de desalocação automática de memória para objetos não mais referenciados, que age em tempo de execução. Alternativamente, pode-se considerar a memória do computador infinita e que não há desalocação de memória. O que interessa são somente as propriedades conceituais, e não os detalhes de implementação de interpretadores ou compiladores que executam uma linguagem de programação em um computador físico.
- Um bloco de código poderá conter blocos aninhados de forma hierárquica.
- Cenários de uso de reflexão computacional intercessiva, que permite que o programa altere dinamicamente a si mesmo, serão ignorados, o que significa que
  - O programa e os dados estão conceitualmente em áreas separadas da memória, sendo que a área do programa é disponível somente para leitura, e a área dos dados é disponível para leitura e escrita.
  - Todos os tipos são definidos estaticamente. A tipagem estática, que possibilita a detecção precoce de erros de tipagem, é uma ferramenta fundamental na busca por melhor confiabilidade dos programas (Meyer, 1997). Assim, não serão feitos comentários sobre linguagens com tipagem dinâmica, inexistente ou não verificada estaticamente.
  - Todo objeto é uma instância existente em tipo de execução de um tipo de dados definido estaticamente.
- Para efeito de simplificação da explicação, também não serão considerados cenários de processamento concorrente, por exemplo com processos ou com *threads*. Será considerada somente computação sequencial.

As abordagens tratadas nesta seção são as seguintes: Paradigma Pré-Procedimental (Subseção 2.4.1), Paradigma Procedimental (Subseção 2.4.2), Paradigma Orientado a Objetos Parcial (Subseção 2.4.3), Paradigma Orientado a Objetos Parcial com Genericidade (Subseção 2.4.4), Paradigma Orientado a Objetos Parcial com Herança (Subseção 2.4.5) e Paradigma Orientado a Objetos Completo (Subseção 2.4.6). Em seguida, na Subseção 2.4.7 são apresentadas algumas conclusões sobre como os paradigmas parecem evoluir historicamente.

2.4.1 Paradigma Pré-Procedimental (PPP)

O Paradigma Pré-Procedimental (PPP) considera o programa como um bloco sequencial e finito de instruções que manipulam objetos (dados). O bloco do programa pode conter alguns blocos aninhados, que determinam escopos. Um objeto é uma representação computacional de uma informação significativa ao usuário que pode ser processada pelas instruções de um programa. O fluxo de controle do programa é determinado por estruturas de sequência, seleção, repetição e ocasionalmente de desvio (*goto*). O programa termina quando aparece uma instrução explícita de terminação, ou quando não há mais instruções a serem processadas.

Neste paradigma, as variáveis possuem as seguintes propriedades fundamentais (Sebesta, 2003):

- Endereço ou valor-e: identidade única da variável para o programa, *independente de escopo*. Esta é a única propriedade que conceitualmente duas variáveis contemporâneas (que coexistem em um subconjunto do tempo de vida de ambas) jamais podem compartilhar.
- Identificador: nome que identifica a variável *em um dado escopo*.
- Tipo (de dado): característica que define os possíveis objetos que a variável pode receber. O tipo especifica a classe de valores que podem ser atribuídos à variável. Neste paradigma, uma variável pode se referir somente a objetos do mesmo tipo que ela, ou ao valor especial nulo.
- Conteúdo, valor ou valor-d: indicação do objeto que a variável referencia. O conteúdo de uma variável é sempre uma referência para um objeto específico, ou um valor especial, denominado *nulo*, que indica referência para nenhum objeto.
- Tempo de vida: intervalo de tempo durante o qual a variável existe para o programa. Neste paradigma, o tempo de vida de uma variável se inicia com a execução de sua declaração e seu término coincide com a conclusão da execução das instruções contidas no bloco que contém a declaração da variável.
- Escopo: trecho da execução do programa no qual a variável é visível e pode ser potencialmente acessada. Uma variável pode ser visível ou acessível somente em um subconjunto do programa durante seu tempo de vida. Por exemplo, em um programa estruturado em blocos aninhados, uma variável de um bloco externo pode se tornar invisível em um bloco interno porque no bloco interno se declarou outra variável (de endereço necessariamente diferente) com o mesmo identificador. Mas uma variável jamais pode ser acessada antes de seu tempo de vida ter se iniciado ou após seu tempo de vida ter expirado.

Uma variável pode passar a referenciar outro objeto somente por meio de uma operação de *atribuição*. Convencionalmente, o lado esquerdo de uma expressão de atribuição indica a variável

que terá um valor atribuído a ela, enquanto o lado direito consiste em uma expressão que, quando executada, retorna o valor que será atribuído.

A diferença básica entre uma linguagem no paradigma pré-procedimental e a linguagem de montagem de um computador físico convencional é que a primeira oferece expressividade suficiente para descrever fórmulas e expressões completas como se cada uma delas fosse uma instrução composta de instruções elementares. A tradução de uma implementação no PPP para linguagem de montagem envolveria uma etapa de tradução de fórmulas.

Existem somente tipos e instruções pré-disponibilizadas (*built-in*), o que significa que não é permitido ao desenvolvedor estender a linguagem com seus próprios tipos e operações. O problema com essa configuração é que dificilmente uma linguagem de programação de propósito geral poderia possuir abstrações embutidas suficientes para todos os usos necessários. Obviamente, o desenvolvedor reusa as abstrações necessárias que existem, mas é importante que uma linguagem forneça mecanismos para que o desenvolvedor crie as abstrações que não existem mas são necessárias. Muitas vezes o desenvolvedor cria uma sequência de instruções de nível mais baixo (disponibilizadas pela linguagem) que poderiam ser agrupadas para representar uma instrução parametrizável de nível mais alto se, na melhor hipótese, houvesse a instrução na linguagem pronta para uso, ou se ao menos houvesse uma maneira do desenvolvedor estender a linguagem para "simular" que aquela nova instrução sempre esteve embutida na linguagem.

A replicação de código piora a evoluibilidade de um programa, já que, caso parte de uma implementação sujeita à replicação tenha de ser alterada por algum motivo, todas as suas réplicas também terão de ser alteradas, uma tarefa entediante e propensa a erros. Há um tipo de replicação de código, frequente em vários programas, que não pode ser solucionado de forma satisfatória utilizando somente os mecanismos presentes no PPP. Observe o exemplo a seguir.

```
//conjunto de instruções A
instrA1;
goto call_X_from_A;
return_to_A: instrA2;

...

//conjunto de instruções B
instrB1;
goto call_X_from_B;
return_to_B: instrB2;

...

call_X_from_A:
instr1; //início da replicação
instr2;
goto return_to_A; //variabilidade da replicação / fim da replicação

...

call_X_from_B:
instr1; //início da replicação
instr2;
goto return_to_B; //variabilidade da replicação / fim da replicação
```

Perceba que o padrão de replicação segue a forma "Desvio para o código replicado, execução do código replicado, desvio para a instrução imediatamente após o desvio original; outro desvio para o código replicado, execução do código replicado, desvio para a instrução imediatamente após o desvio original". Algum mecanismo é necessário para modularizar a implementação replicada e tornar o desvio final independente de uma posição fixa, já que a única variabilidade que envolve a replicação é a posição do retorno do desvio informada ao fim do código replicado.

Como se percebe, a estrutura de um programa no PPP é bastante rudimentar. Uma linguagem imperativa de alto nível que segue o PPP contém somente o mínimo de recursos necessários para se criar um programa em uma linguagem imperativa de alto nível. Isso está longe de ser adequado para desenvolver programas de forma minimamente profissional utilizando os princípios de Engenharia de Software moderna.

2.4.2 Paradigma Procedimental (PP)

As deficiências do PPP sugerem que deveria haver algum mecanismo que possa ser utilizado pelo desenvolvedor para estender a linguagem de programação com novas instruções ou operações. A partir do Paradigma Procedimental (PP), isso é possível, por meio de um mecanismo de modularização denominado *rotina*.

Para definir precisamente tal mecanismo de modularização, é necessário verificar qual é a variabilidade na replicação do código que não podia ser tratada no PPP. No caso, a única variabilidade ocorre ao fim, que é registrar o ponto de retorno do desvio. Isso pode ser feito implicitamente pelo

sistema em tempo de execução por meio de uma *pilha de execução*. Quando o desenvolvedor invoca uma rotina, é colocado na pilha de execução o endereço da instrução imediatamente após a invocação original. Dessa forma, o desvio pode ser feito para o ponto correto, e não há necessidade de mencioná-lo explicitamente, como era feito no PPP.

Uma rotina possui somente um ponto de entrada e, em geral, um ponto de saída. Para a execução entrar na rotina é necessário chamá-la ou invocá-la. A saída de uma rotina, denominada retorno, sempre é direcionada ao ponto imediatamente após o ponto no qual ela foi anteriormente chamada, que é registrado na pilha de execução. Um exemplo de invocação e de definição de uma rotina é mostrado a seguir.

```
inicio do programa
    int a, b, c, menorValor;

    imprima "Forneça três números";
    leia a, b, c;
    menorValor := menorDeTres( a, b, c );
    imprima "O menor dos três números fornecidos é: ", menorValor;
fim do programa

menorDeTres( int n1, int n2, int n3 ): int
inicio
    int menor;

    se n1 < n2
        menor := n1;
    senão
        menor := n2;

    se n3 < menor
        menor := n3;

    retorne menor;
fim
```

Em um contexto no qual existem rotinas, as variáveis do programa podem ser declaradas fora ou dentro das rotinas. Além disso, aparecem restrições no acesso a variáveis; por exemplo, a variável de uma rotina não pode ser diretamente acessável por outra rotina. Com isso surgem novas restrições de *escopo*. O tempo de vida de variáveis declaradas fora de rotinas termina juntamente com o término da execução do programa, significando que o programa é um bloco que contém outros blocos (as rotinas também definem um bloco).

Por conveniência, o PP define outro tipo de módulo, o *registro*. Um registro é um tipo estruturado incompleto, cujo propósito é abstrair a definição de estruturas de dados complexas, e portanto não encapsula a implementação de instruções. Ele é incompleto porque, conforme será abordado no Paradigma Orientado a Objetos Parcial, sua definição não indica quais operações são permitidas naquele tipo. De fato, qualquer operação definida pelo desenvolvedor pode acessar sem restrições a representação subjacente de um registro. Por este motivo, o registro abstrai a definição de estruturas de dados, mas não abstrai adequadamente seu uso.

Na prática, as rotinas são tão frequentemente utilizadas no PP que os programas acabam sendo efetivamente estruturados com base em rotinas. Essa é a principal diferença entre o *design* de um programa no PPP e o *design* de outro programa equivalente em *PP*.

Os mecanismos disponibilizados pelo PP permitem a criação de uma biblioteca de rotinas reutilizáveis. Tais bibliotecas são adequadas quando as seguintes limitações são satisfeitas (Meyer, 1997):

- É possível caracterizar cada problema por um pequeno conjunto de argumentos de entrada e de saída;
- Os problemas são claramente distintos uns dos outros;
- Estruturas de dados complexas não estão envolvidas, pois caso contrário seria necessário distribui-las entre as rotinas, assim prejudicando a autonomia conceitual de cada módulo.

Essas limitações gerais do PP sugerem que há problemas que poderiam ser melhor solucionados utilizando novos mecanismos de modularização. Por exemplo, o PP possui uma deficiência séria no tratamento de estruturas de dados complexas. A existência em um programa de várias rotinas, muitas vezes não relacionadas logicamente, que acessam a representação subjacente de objetos de um tipo de registro é prejudicial à evoluibilidade do programa. Se por algum motivo for necessário alterar a representação subjacente de um registro, todas as rotinas que processam objetos daquele tipo terão de ser modificadas invasivamente. Portanto, é necessário algum mecanismo que permita tornar a representação subjacente de um registro independente do processamento de objetos de um determinado tipo.

2.4.3 Paradigma Orientado a Objetos Parcial (POOP)

No Paradigma Orientado a Objetos Parcial (POOP), a definição de tipo é alterada para facilitar a evolução dos registros, e isso por consequência afeta também a estruturação das rotinas. A motivação para o POOP é que um tipo não é definido pela sua representação subjacente, mas pelas operações que agem sobre objetos daquele tipo.

Para compreender porque rotinas / operações são necessárias na definição dos tipos, considere o que ocorre ao se enxergar um tipo somente como um registro. Todos os programas que usam o tipo seriam implementados em termos da representação subjacente do registro. Contudo, ao mudar a implementação do tipo por algum motivo, todos os programas que utilizam o tipo devem ser alterados. Por outro lado, se as rotinas / operações que manipulam o tipo forem definidas como parte da definição do tipo, somente tais rotinas acessariam a representação subjacente do tipo, enquanto o restante do programa acessaria a implementação do tipo apenas indiretamente, por meio das rotinas. Caso seja necessário alterar a implementação subjacente do tipo, somente as rotinas que são parte da definição do tipo teriam de ser alteradas; o restante do programa permaneceria

intacto, desde que a interface do tipo se mantivesse. Se rotinas suficientes forem fornecidas, a falta de acesso à representação não causará aos usuários do tipo nenhuma dificuldade, pois qualquer coisa que eles quiserem que seja feita com os objetos poderá ser feita eficientemente chamando rotinas do tipo. Assim, atinge-se abstração por especificação para tipos tornando as rotinas parte da definição do tipo (Liskov e Guttag, 1986).

Essa nova perspectiva de tratamento de tipos é o fundamento da teoria dos Tipos Abstratos de Dados (TADs). O conceito de TAD é em parte consequência do princípio de *ocultação de informação*: TADs bem desenhados escondem detalhes de representação subjacente e permitem a manipulação de suas instâncias abstratas somente por meio de rotinas de nível mais alto (Parnas, 1972).

Escolher estruturas de dados (representações subjacentes de tipos) adequadas é crucial para obter um programa eficiente. Na ausência da abstração de dados, as estruturas de dados devem ser definidas muito precocemente, antes da implementação dos módulos que as usam serem desenhados. Neste ponto, contudo, os usos dos dados tipicamente não são bem compreendidos. Portanto, pode estar faltando na estrutura escolhida uma informação necessária ou ela pode estar organizada de maneira ineficiente. Uma vantagem do POOP é o fornecimento de mecanismos para impedir o acesso direto à representação subjacente de um tipo para reduzir problemas de evolução. Registros não são mais necessários, pois podem ser simulados por uma classe. A abstração de dados permite deferir decisões sobre estruturas de dados até que os usos dos dados estejam completamente compreendidos. Em vez de se definir a estrutura imediatamente, introduz-se um tipo abstrato de dados com seus objetos e operações válidos. As implementações dos módulos clientes podem então ser desenhadas em termos do tipo abstrato. Devisões sobre como implementar o tipo são feitas mais tarde, quando todos os seus usos são compreendidos (Liskov e Guttag, 1986).

A abstração de dados é também valiosa durante a evolução do programa. Nesta fase, é provável que seja necessário alterar as representações subjacentes de alguns tipos para melhorar o desempenho ou acomodar requisitos novos. A abstração de dados ajuda a limitar as mudanças para somente a implementação dos tipos; nenhum dos módulos clientes precisaria ser alterado (Liskov e Guttag, 1986).

Já que no Paradigma Orientado a Objetos Parcial objetos de um tipo são usados somente por meio de chamadas a rotinas do tipo, a maior parte da especificação de um tipo consiste em explicar o que suas rotinas fazem. A definição de tipo para o Paradigma Orientado a Objetos Parcial inclui a definição de tipo do PP e a estende afirmando que a implementação de um tipo de dados corresponde a uma nova espécie de módulo, denominada *classe*. A partir do POOP, há o mecanismo de classe para que um desenvolvedor possa estender a linguagem de programação com novos tipos.

Assim, no POOP a definição de tipo torna-se mais complexa e poderosa. Um tipo de dado passa a ser definido como a característica que define os possíveis objetos que a variável pode receber (definição antiga), *bem como as operações válidas para manipular tais valores* (extensão da definição).

Por si só, o ato de descobrir que tipos são definidos por conjuntos de rotinas não acarreta uma mudança de paradigma. Mas o ato consciente do programador de estruturar o programa por meio de classes em vez de rotinas acarreta uma mudança de paradigma real, pois muda radicalmente a estrutura do *design* de um programa comparado com o *design* de um programa equivalente em PP, e muda também concepção sobre o que significa um programa bem desenhado.

2.4.4 Paradigma Orientado a Objetos Parcial com Genericidade (POOPG)

O Paradigma Orientado a Objetos Parcial com Genericidade (POOPG) é uma evolução do POOP que adiciona mecanismos para resolver um problema específico de replicação de código.

Suponhamos que é necessário em vários vários programas, criar pilhas de objetos de determinados tipos de dados. Em um programa, deve-se criar uma pilha de inteiros; em outro programa, uma pilha de *strings* é importante; em ainda outro programa, uma pilha de cores contribuiria para completar a implementação do programa. Ao comparar diretamente as diferentes implementações de pilha, é fácil observar que as implementações são muito similares, o que evidencia replicação indesejada de código. A única variabilidade entre as diferentes implementações de pilha é o tipo de dado dos objetos que podem ser inseridos nela. Seria então interessante se pudéssemos implementar a pilha somente uma vez e ela pudesse ser parametrizada para diferentes tipos de dados à medida que novos tipos fossem necessários. O POOPG fornece exatamente esta capacidade. A partir do POOPG, é possível criar classes genéricas, isto é, classes parametrizáveis.

Genericidade é a possibilidade de criar e utilizar classes parametrizáveis. O POOPG considera que todo tipo é parametrizável. Essa propriedade implica que um tipo convencional no POOP é simplesmente um tipo parametrizável sem parâmetros do POOPG.

A idéia de genericidade não foi criada no POOPG; a genericidade é utilizada informalmente desde o PPP, quando se declaravam *arrays* de tipos diferentes. Mas é só a partir do POOPG que o conceito de genericidade é definido de forma mais explícita e rigorosa.

A genericidade é uma facilidade para o *fornecedor* de módulos, isto é, para o desenvolvedor de classes reusáveis. Ela possibilita localizar em um módulo a implementação de um determinado conceito, aplicado a diferentes tipos. Isso resolve a questão da variação de tipos (Meyer, 1997).

2.4.5 Paradigma Orientado a Objetos Parcial com Herança (POOPH)

O Paradigma Orientado a Objetos Parcial com Herança (POOPH) é uma evolução do POOP que adiciona mecanismos para resolver um problema específico de replicação de código, diferente e ortogonal ao problema tratado pelo POOPG.

O POOPH parte da descoberta de que tipos podem ser categorizados, isto é, tipos podem ser subtipos ou supertipos de outros tipos. Por exemplo, os tipos Mapa (ou Dicionário), Conjunto e *Array* podem ser todos considerados subtipos do tipo Coleção. O mecanismo de herança foi criado com o propósito de criar subtipos a partir de um tipo-base, permitindo assim maior capacidade de

abstração (pela categorização de tipos em hierarquias de generalização), maior possibilidade de reuso e menor replicação de código.

Uma propriedade da herança é que todas as rotinas válidas para um supertipo são automaticamente válidas para todos os subtipos. Essa facilidade melhora bastante a reusabilidade de módulos em POOPH. O mecanismo de herança também pode ser usado para reduzir o fenômeno de replicação de código. Quando um código aparece replicado em várias classes, frequentemente é possível localizar este código em uma superclasse comum a todas as classes envolvidas na replicação. Assim, este código passa automaticamente a pertencer a todas as subclasses.

A partir do POOPH, se uma variável não estiver referenciando o valor nulo, o tipo do objeto que a variável referencia é sempre igual a ou um subtipo do tipo da variável. Isso significa que surge em POOPH o conceito de *variáveis polimórficas*. Na linguagem de programação *Java*, por exemplo, uma variável polimórfica do tipo *Object* pode receber objetos originados de quaisquer classes, pois a classe *Object* é a classe de nível mais alto na hierarquia de classes de qualquer programa. Contudo, o objeto referenciado por uma variável da classe *Object* pode estar sujeito apenas a ser passado como parâmetro de rotinas que aceitem objetos da classe *Object*, independentemente de qual é o tipo real do objeto envolvido.

2.4.6 Paradigma Orientado a Objetos Completo (POO)

O Paradigma Orientado a Objetos Completo (POO) corresponde à junção harmoniosa dos conceitos de herança e genericidade em uma abordagem. Pode-se considerar que *POO = POOPG + POOH*, ou, alternativamente, que *POO = POOP + Genericidade + Herança*. O POO é o melhor conjunto de mecanismos possível para melhorar a compreensibilidade, a reusabilidade e a evoluibilidade do programa que considera a classe como o único módulo de primeira ordem existente.

A Figura 2.1 mostra um esquema que combina o uso de herança e de genericidade para obter código reusável para vários cenários. Supomos que já existem de antemão as classes Livro, Inteiro e *String*. Se as classes da primeira coluna forem implementadas, é possível obter automaticamente objetos correspondentes às classes das demais colunas.

2.4.7 Como os paradigmas evoluem

A discussão feita neste capítulo evidencia que os paradigmas de programação imperativa têm historicamente se desenvolvido sobre os conceitos erigidos pelo paradigma anterior a partir da observação de um ou mais dos seguintes pontos:

- **Identificação de problemas de *design***. Inicialmente, tenta-se encontrar uma maneira de solucionar um problema de *design* de um programa por meio dos construtos que o paradigma e a linguagem de programação fornecem. Caso um problema seja recorrente, a descrição do

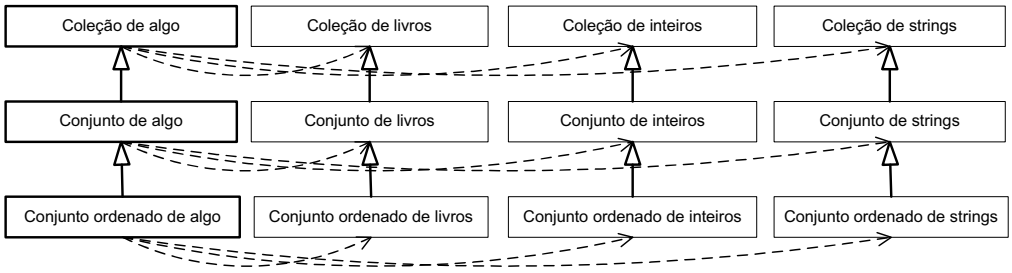


Figura 2.1: Exemplo de combinação de herança e genericidade. As setas contínuas simbolizam herança, enquanto as setas seccionadas simbolizam uso de genericidade.

problema e a solução mais satisfatória possível para aquele problema utilizando o paradigma são documentados em um padrão (*pattern*). Note que a solução mais satisfatória pode ainda assim não ser plenamente satisfatória, o que sugere uma deficiência inerente ao paradigma.

- **Identificação correta de algumas deficiências inerentes ao paradigma que tornam as mudanças inevitavelmente invasivas**. A existência de deficiências que são supostamente tidas como inerentes motiva os pesquisadores a desenvolverem novos construtos para minimizar tais problemas. Uma deficiência comum é a replicação inevitável de código em algum contexto recorrente em programas distintos, e não somente de forma interna a um único programa, quando se utiliza um determinado paradigma. Essa deficiência em particular foi o que motivou a criação da rotina para o paradigma procedimental e a criação de herança no paradigma orientado a objetos. Outra deficiência comum é a detecção de elementos do programa que são logicamente relacionados mas que não estão separadamente modularizados (isto é, explicitamente isolados) de outros elementos. Essa separação não deve ser somente sintática, como divisão do programa em pacotes ou arquivos diferentes; o novo módulo que irá conter tais elementos deve possuir uma semântica especial que afeta a maneira de efetuar o *design* do programa. No caso do paradigma orientado a objetos, outra deficiência encontrada foi o suporte ruim que o paradigma procedimental fornecia para tratar dados, que são elementos muito importantes de um programa, já que dos principais objetivos de um programa é processar e manipular dados que muitas vezes são complexos.

- **Desenvolvimento de uma nova maneira de abstrair partes do programa e possibilitar a extensão da linguagem de programação pelo desenvolvedor**. Isso ocorreu antes do PP quando se percebeu que determinadas sequências de instruções que se repetiam na implementação poderiam ser substituídas por uma instrução de nível mais alto. Isso ocorreu antes do POOP quando se percebeu que as operações que agem sobre objetos de um tipo são parte da definição daquele tipo.

A partir dessas observações, serão analisados na próxima seção e no próximo capítulo as deficiências do POO e serão discutidos alguns mecanismos criados para superar tais deficiências.

2.5 Abordagens Imperativas Recentes

Uma categorização possível para as abordagens tratadas nesta seção é conforme o nível de simetria, definida como a existência de somente um tipo de módulo de primeira ordem. A abordagem assimétrica de implementação *AspectJ* (The AspectJ Team, 2006) divide o *design* do programa em uma base, estruturada por meio de classes, e (possivelmente) vários aspectos que afetam aquela base. As abordagens simétricas de implementação *Hyper/J* (Tarr et al., 1999; IBM Research, 2000) e *Concern Manipulation Environment* (CME) (The CME Team, 2005) dividem um modelo de *design* em hiperfatias ou módulos de interesses estruturais.

Esta seção se divide da seguinte maneira. A Subseção 2.5.1 contém uma discussão sobre algumas deficiências do POO. Depois, discute-se sobre as abordagens de Desenvolvimento de Software Orientado a Aspectos com *AspectJ* (Subseção 2.5.2) e *MDSoC/Hyperspaces* com *Hyper/J* (Subseção 2.5.3).

2.5.1 Deficiências do Paradigma Orientado a Objetos

O paradigma orientado a objetos completo certamente não é a solução final para o problema de modularização de software, já que ele também possui suas próprias deficiências. À medida que essas deficiências vêm sendo melhor compreendidas, algumas abordagens têm surgido para complementar o paradigma orientado a objetos completo com novas possibilidades de modularização. Algumas dessas deficiências são apontadas na discussão que se segue.

A criação de subclasses para estender não-invasivamente a funcionalidade de uma classe frequentemente não é satisfatória, porque requer mudanças invasivas aos clientes que utilizavam a classe original e também porque a criação de subclasses pode acarretar explosão combinatória do número de classes (Tarr et al., 1999). Deveria haver algum mecanismo para inserir rotinas e atributos em uma classe de maneira não-invasiva.

O problema de limitar o impacto de mudança tem sido atacado no POO por várias arquiteturas e mecanismos, como os mencionados por Tarr et al. (1999): invocação implícita, mediadores, integração baseada em eventos, padrões de *design* e outros. Todos eles são valiosos, mas sofrem do problema de que as mudanças que eles permitem (os pontos abertos) devem ser antecipados. Tais mecanismos requerem um pré-planejamento significativo do *design*. No POO, inserir quaisquer destes mecanismos em locais onde eles não foram originalmente planejados requer mudanças invasivas.

Outra deficiência em POO é a existência de trechos de código relacionados que estão espalhados por várias classes ou por várias rotinas. Observe o exemplo exposto na Figura 2.2. Algumas rotinas de classes diferentes implementam também código relativo à funcionalidade de *logging*. O melhor que se pode fazer para amenizar essa replicação de código é criar uma superclasse que contém rotinas para *logging*, como demonstrado na Figura 2.3, mas de qualquer forma as rotinas das subclasses devem explicitamente chamar as rotinas da superclasse nos momentos corretos. Se

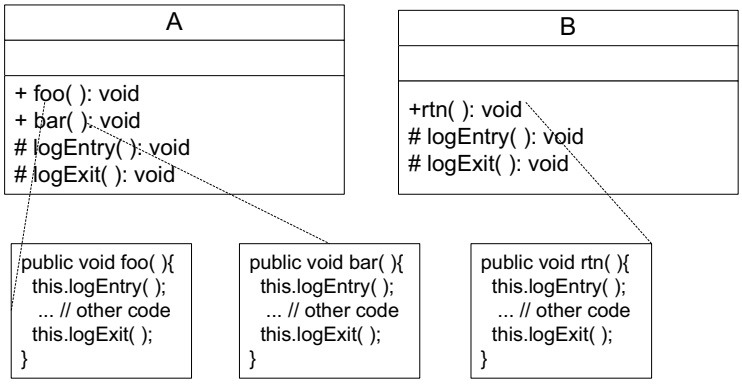


Figura 2.2: Rotinas de classes diferentes que implementam *logging*.

não se desejar mais a funcionalidade de *logging*, as chamadas de rotinas da superclasse espalhadas pelas subclasses terão de ser todas removidas.

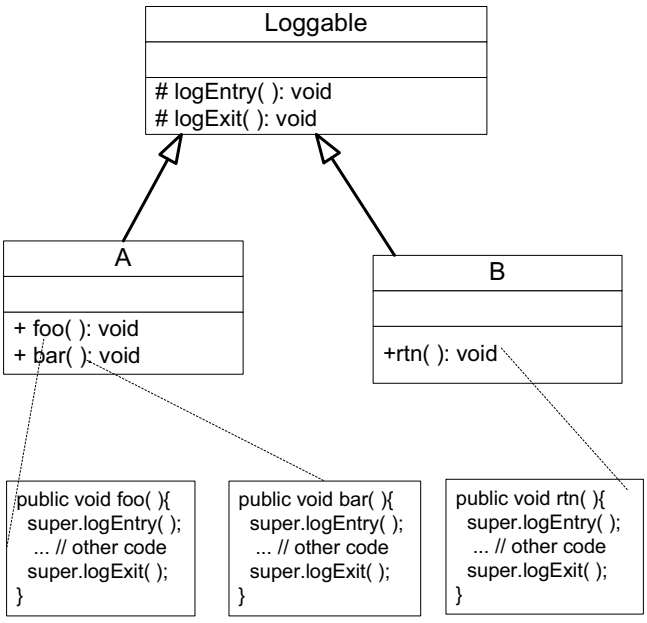


Figura 2.3: Transferência das rotinas de *logging* para uma superclasse comum.

Um ponto em comum entre as abordagens imperativas recentes aqui tratadas é o reconhecimento de que os maiores problemas do POO são causados em grande parte por limitações inerentes ao paradigma na modularização separada da implementação relativa a um tipo de interesses denominado aqui *estrutural*. Muitas mudanças afetam a implementação relativa a interesses es-

truturais específicos, que não se alinham à estrutura de classes. Se interesses estruturais puderem realmente ser separados de uma forma adequada, seria possível reduzir gargalos e aumentar o paralelismo do processo de desenvolvimento/evolução, já que módulos diferentes poderiam ser alocados a membros distintos da equipe que trabalhariam simultaneamente. Como conseqüência, o tempo de liberação do produto se reduziria.

A presença de implementação não-modularizada de interesses estruturais causa dois efeitos sintomáticos associados na estrutura do sistema (Kiczales et al., 1997). Primeiramente, no ponto de vista do sistema, o problema do *espalhamento* ocorre: a implementação de cada interesse estrutural está possivelmente espalhada sobre muitos módulos, ao invés de estar localizada em um único módulo. Em segundo lugar, tomando o ponto de vista de um módulo afetado, o problema do *entrelaçamento* é observado: parte da implementação geral de um interesse estrutural está entrelaçada em um mesmo módulo com parte da implementação de outro interesse estrutural, ao invés de estar em um local separado. O espalhamento contribui para aumentar o acoplamento entre interesses estruturais, enquanto o entrelaçamento contribui para reduzir a coesão (Kaminski, 2001). Isso significa que a modularização do programa por meio de classes (que contêm a implementação de tipos abstratos de dados) não se alinha ao escopo das mudanças que envolvem interesses estruturais.

Note que, como cada módulo deve conter a implementação de somente um interesse estrutural, aquela implementação não deve ser poluída pela (entrelaçada com a) implementação de outros interesses estruturais. Em outras palavras, um módulo que implementa um interesse estrutural não deve ter consciência (isto é, deve ser *oblivious*) (Filman e Friedman, 2000), tanto quanto possível, da existência de outros interesses. O cenário ideal para todo módulo é máxima coesão interna e mínimo acoplamento a outros módulos.

Assim, deve ser feita também a modularização do programa por meio de módulos que contêm a implementação de interesses específicos. A maneira de efetuar tal modularização é a principal diferença entre as abordagens tratadas a seguir.

2.5.2 Desenvolvimento de Software Orientado a Aspectos (DSOA) com *AspectJ*

A abordagem de *AspectJ* para o Desenvolvimento de Software Orientado a Aspectos parte do princípio que em uma implementação certos interesses estruturais são primários às classes, e portanto devem ser mantidos nelas, enquanto outros interesses estruturais são secundários ou menos fundamentais. Contudo, em POO não há saída a não ser manter a implementação dos interesses estruturais secundários espalhadas sobre várias classes e entrelaçadas com a implementação dos interesses estruturais primários em cada classe. O fato de que a implementação relativa a interesses estruturais secundários se espalha por várias classes parece indicar que tal implementação não deveria pertencer a elas, mas a outro tipo de módulo. A implementação dos interesses estruturais secundários é importante, mas é indesejável que ela esteja misturada à implementação das

classes. Deve haver, então, alguma maneira de separar a implementação dos interesses estruturais secundários da implementação das classes (Kiczales et al., 1997).

Assim, na abordagem de DSOA com *AspectJ* modulariza-se um programa por meio de uma separação explícita entre interesses estruturais primários, que são implementados em um conjunto de classes (a *base*) da mesma maneira que no POO, e interesses estruturais secundários, que devem ser implementados em módulos denominados *aspectos*. Em *AspectJ*, o desenvolvedor cria uma base para o programa e em seguida cria aspectos separadamente codificados para aumentar as funcionalidades das classes e rotinas. Isso é especialmente poderoso quando um único aspecto entrecorta muitas classes, permitindo uma implementação localizada de um comportamento espalhado (Ossher e Tarr, 2001).

O conceito de *ponto de junção* é central na abordagem de *AspectJ*. Pontos de junção são elementos da semântica do programa que podem ser afetados por aspectos. Os pontos de junção não são necessariamente construtos explícitos na linguagem de programação usada para implementar a base. Eles podem ser elementos implícitos na semântica do programa, dependentes de informações obtidas durante a execução (Kiczales et al., 1997).

A linguagem usada para implementar os aspectos fornece módulos e mecanismos de interação intermodular diferentes daqueles fornecidos pela linguagem usada para implementar a base. Um processador especial chamado *combinador aspectual* é usado para coordenar a composição entre a base e os aspectos, utilizando para isso informação sobre os pontos de junção (Kiczales et al., 1997).

O modelo de execução de um programa em *AspectJ* é baseado na interceptação ocasional do fluxo de controle *default* do programa. Em um nível alto, ele funciona da seguinte forma: Em um programa P, tão logo a condição C aconteça, efetue a ação A (pertencente ao aspecto) (Filman e Friedman, 2000). O mecanismo de *invocação implícita*, ativado quando a condição observada é verdadeira, é uma vantagem para implementar programas de alta complexidade. Os especialistas no domínio da aplicação provavelmente não são familiarizados com os detalhes dos algoritmos especializados para interesses como distribuição, autenticação, controle de acessos, sincronização, encriptação, redundância e outros. Portanto, nem sempre se pode confiar que eles implementarão a invocação dos comportamentos apropriados nos momentos corretos, especialmente em um cenário de evolução do programa, no qual ele está sujeito a mudanças frequentes (Elrad et al., 2001).

Condições baseadas em informações obtidas em tempo de execução frequentemente dependem do uso de reflexão. Kiczales et al. (1997) afirmam que o DSOA possui uma conexão profunda com a reflexão computacional. Neste sentido, *AspectJ* pode ser considerado também uma ferramenta que incorpora uma forma mais disciplinada e talvez até mais eficiente (em tempo de execução) de usar reflexão a favor de uma boa modularização do programa. Filman e Friedman (2000) menciona várias informações que podem ser obtidas em tempo de execução, como: o lançamento de uma exceção; uma chamada de uma rotina X no escopo temporal de uma chamada a Y; o tamanho da pilha de execução; padrões na história do programa (por exemplo, após a rotina de entrada de senha ter falhado cinco vezes, sem nenhum sucesso entre tais execuções).

Nesta abordagem também se discute o conceito de *interesses entrecortantes*, que, grosso modo, são interesses estruturais cujas implementações entrecortam vários outros módulos do programa (classes e rotinas). Já que a meta que se deseja atingir é "um módulo, um interesse estrutural", a abordagem de *AspectJ* assume que todo módulo implementa um interesse estrutural "central" que descreve sua essência. Os interesses entrecortantes são outros interesses estruturais acidentais que, por alguma razão, estão entrelaçados com a implementação do interesse estrutural "central" daquele módulo. Assim, os interesses entrecortantes são os interesses estruturais cujas implementações tendem a se entrelaçar e a se espalhar por diversos módulos quando se usam técnicas de programação tradicionais (como orientação a objetos). Contudo, ainda não há consenso quanto à definição precisa e ao conjunto de propriedades de interesses entrecortantes, como se pode perceber em discussões recentes na lista de DSOA[1].

O conceito de interesse estrutural difere do conceito de interesse entrecortante porque o último é considerado não-pertencente aos módulos em que a implementação relativa a ele aparece entrelaçada com a implementação relativa a outros interesses. A implementação relativa a um interesse estrutural, ao contrário, é considerada pertencente àqueles módulos em uma determinada dimensão, mas também pertence a um módulo de outro tipo quando uma outra dimensão de visualização é adotada.

O fato de que a base dita a estrutura do programa traz conveniências e limitações à especificação de aspectos. É difícil fazer com que aspectos explicitamente aumentem uns aos outros, pois eles são especificados em função da base. Além disso, em cenários de reuso e integração, geralmente não há uma única referência compartilhada por todos os componentes: componentes reusáveis desenvolvidos separadamente empregam os modelos mais apropriados para seus propósitos específicos, assim eles devem ser adaptados ao contexto de reuso, e diferenças entre componentes sendo integrados devem ser reconciliadas. Mas os aspectos tipicamente estão vinculados a uma base específica, o que pode limitar sua reusabilidade e dificultar sua compreensão quando não se analisa também a base (Ossher e Tarr, 2001).

Há uma faixa de simetria de tipos de entrelaçamento, contida entre os dois extremos a seguir:

- *Entrelaçamento simétrico*: não há interesse estrutural "dominante"; ambos são igualmente importantes para justificar a existência da rotina. Diz-se que tais interesses se *sobrepõem em uma rotina*.

- *Entrelaçamento assimétrico*: para a rotina, um interesse estrutural é essencial, enquanto o outro tem um papel mais periférico. Diz-se que o interesse estrutural menos essencial é *estritamente entrecortante* ao outro interesse, chamado *base*.

O problema com a filosofia do *AspectJ* de separação entre elementos primários e elementos secundários a um módulo é que frequentemente é arbitrário efetuar uma divisão entre o que é

[1] `http://aosd.net/pipermail/discuss_aosd.net/2006-February/thread.html`

essencial a uma rotina de uma classe e o que é acidental a ela, porque em alguns casos não está claro o que é essencial e o que é acidental. Quanto mais simétrico for o entrelaçamento, mais pronunciado será esse problema. Assim, a utilização do *AspectJ* parece ser mais adequada quando a "essência" e a "parte acidental" de uma classe ou de uma rotina são facilmente determinadas, isto é, quando o entrelaçamento tende a ser mais assimétrico.

A linguagem *AspectJ* é uma extensão da linguagem de programação *Java* cujo objetivo é permitir a implementação adequada de aspectos (The AspectJ Team, 2006). Basicamente, as novas construções do *AspectJ* consistem em: conjuntos de junção (*pointcuts*) que identificam conjuntos de pontos de junção; adendos (*advice*) que definem o comportamento em um dado conjunto de junção; construções para afetar estaticamente a estrutura dos módulos básicos do programa (declarações intertipos e que alteram a hierarquia de classes); e os aspectos (*aspects*) que encapsulam as construções novas e as tradicionais de uma classe Java. Um *aspecto* é um módulo cujo propósito original é encapsular a implementação relativa a um interesse entrecortante (Kiczales et al., 2001).

Na Figura 2.4 é mostrada uma implementação em *AspectJ* para o exemplo de interesse entrecortante apresentado anteriormente. Essa implementação será utilizada ao longo da seção. O aspecto *AtualizacaoDaTela* se encarrega de atualizar a visualização todas as vezes que alguma figura é alterada. Como as figuras são alteradas a partir dos métodos que alteram suas coordenadas, os adendos são definidos nas chamadas a esses métodos.

```
public class Ponto implements ElementoDeFigura {
  private int x = 0, y = 0;
  int getX() { return x; }
  int getY() { return y; }
  void setX(int x) { this.x = x; }
  void setY(int y) { this.y = y; }
  public void move(int dx, int dy) { x += dx; y += dy; }
}

public class Linha implements ElementoDeFigura {
  private Ponto p1, p2;
  Ponto getP1() { return p1; }
  Ponto getP2() { return p2; }
  void setP1(Ponto p1) { this.p1 = p1; }
  void setP2(Ponto p2) { this.p2 = p2; }
  public void move(int dx, int dy) { p1.move(dx,dy); p2.move(dx,dy); }
}

public aspect AtualizacaoDaTela {
  public pointcut mudancaEstado() :
    call(void ElementoDeFigura.move(int,int)) ||
    call(* Ponto.set*(*)) || call(* Linha.set*(*));

  before(): mudancaEstado() {
    System.out.println("O estado vai mudar...");
  }


  after() returning : mudancaEstado() {
    Tela.atualiza();
  }
}
```

Figura 2.4: Exemplo de código em *AspectJ*

Conjuntos de junção são utilizados para identificar pontos de junção no fluxo do programa. A partir da especificação desses pontos, regras de combinação podem ser definidas – tal como realizar uma certa ação antes ou depois da execução dos pontos de junção. Além disso, os conjuntos de junção podem expor as informações de contexto dos pontos de junção alcançados, podendo ser utilizadas pelas ações que afetam esses pontos. No *AspectJ*, os conjuntos de junção podem conter nomes ou ser anônimos. Na Figura 2.5 é mostrado um exemplo de conjunto de junção nomeado. Um conjunto de junção pode ser definido como uma combinação de conjuntos de junção, utilizando operadores lógicos binários 'e' (&&) e 'ou' (|| – como no exemplo). Além disso, o operador unário de negação (!) também pode ser usado quando não se quer capturar pontos de junção definidos por um conjunto de junção específico. O primeiro conjunto de junção que compõe *mudancaEstado*, por exemplo, captura todas as chamadas ao método *move* da classe *ElementoDeFigura*, que recebe dois inteiros como parâmetros e não retorna nenhum valor.

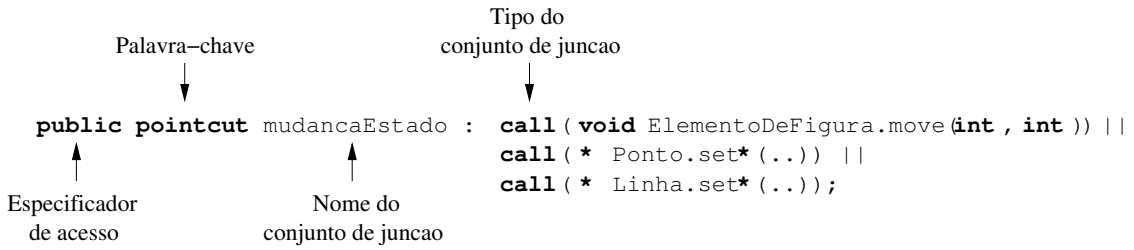


Figura 2.5: Exemplo de definição de conjunto de junção nomeado (adaptado do livro de Laddad (2003))

...

No *AspectJ* especificamente, as principais diferenças entre rotinas e adendos são:

- adendos não podem ser chamados diretamente: a invocação de cada um deles é feita automaticamente pelo ambiente subjacente em tempo de execução;
- adendos são anônimos;
- não se aplicam especificadores de acesso (*public*, *private*, *protected* e outros) a adendos;
- os adendos possuem acessos a outras variáveis especiais além de *this* e *super*: *thisJoinPoint* e *thisJoinPointStaticPart*.

É possível afetar estaticamente o programa por meio de declarações intertipos, modificações da hierarquia de classes, e introdução de avisos e erros de compilação. As declarações intertipos permitem a introdução de novos atributos e rotinas nas classes originais do programa.

As boas idéias do *AspectJ* foram uma fonte de inspiração muito importante para a abordagem descrita nesta dissertação. Foi também valioso aprender com as deficiências do *AspectJ* para evitar a reedição de alguns de seus problemas.

2.5.3 *MDSoC/Hyperspaces* com *Hyper/J* e com o *Concern Manipulation Environment*

É importante que haja uma maneira de visualizar e manipular a implementação relativa a um interesse estrutural separadamente da implementação dos demais interesses estruturais. No POO, contudo, isso não é possível porque só se permite a visualizar a implementação em uma estrutura de classes. A maneira dominante de decompor o programa no POO, a divisão do programa em classes, impõe uma estrutura no software que torna impraticável localizar a implementação de interesses estruturais de forma efetiva. Na dimensão de tipos de dados, a única dimensão de visualização possível no POO, a implementação relativa a interesses estruturais fica espalhada por várias partes do programa modularizadas separadamente umas das outras. Este problema é denominado *tirania da decomposição dominante* (Tarr et al., 1999).

Durante o desenvolvimento do programa, deveria ser possível ao desenvolvedor focar somente as unidades (no caso, partes da implementação de classes) que são pertinentes à tarefa específica e ignorar todas as outras. Para realizar isso, deve-se identificar os interesses estruturais, e se localizar as unidades correspondentes a um interesse estrutural em um só módulo. Idealmente, um desenvolvedor precisaria analisar somente um módulo se ele estivesse interessado em um determinado interesse estrutural (Tarr et al., 1999).

Na abordagem de Separação Multidimensional de Interesses em Hiperespaços (*MDSoC/Hyperspaces* na sigla em inglês), a modularização de software envolve a sua decomposição em uma ou mais dimensões de interesses estruturais, a fim de auxiliar a reduzir a complexidade do sistema, melhorar a compreensão, promover rastreabilidade dentro e através dos artefatos, limitar o impacto das modificações, facilitar a evolução e adaptação não-invasiva e simplificar a integração de componentes. Esses objetivos são difíceis de alcançar no POO porque a separação de interesses em apenas uma dimensão trata de alguns interesses e negligencia outros. Além disso, qualquer critério de decomposição e integração é adequado para alguns contextos e requisitos, mas inadequado para outros (Tarr e Ossher, 2000; Tarr et al., 1999).

MDSoC/Hyperspaces é a evolução da abordagem de *Subject-Oriented Programming* (SOP) (Harrison e Ossher, 1993). O objetivo principal dessa abordagem de Interesses é atacar a Tirania da Decomposição Dominante. O linguagem de programação mais conhecida para a implementação de sistemas utilizando *Hyperspaces/MDSoC* é o *Hyper/J* (Tarr e Ossher, 2000; IBM Research, 2000).

O módulo que contém a implementação de um interesse estrutural é denominado *hiperfatia*. A implementação interna de uma hiperfatia é formada por implementações parciais de classes, que constituem somente a implementação correspondente ao interesse estrutural tratado pela hiperfatia. Em outras palavras, uma hiperfatia é formada por uma estrutura de classes na qual cada classe contém somente o subconjunto de rotinas e atributos pertinentes ao interesse estrutural tratado (Tarr e Ossher, 2000). Isso significa que estes módulos podem não satisfazer todas as restrições de completeza que o POO normalmente requer (Tarr et al., 1999).

A fim de desenvolver o software utilizando *Hyper/J*, o desenvolvedor é responsável por aplicar especialização no domínio do problema, conhecimento na tecnologia de desenvolvimento (especialmente na abordagem de modularização e em ferramentas de apoio relacionadas), e um processo disciplinado para (Tarr e Ossher, 2000):

- identificar interesses estruturais;
- encapsular a implementação de cada interesse estrutural em uma hiperfatia;
- integrar todas as hiperfatias para formar um sistema útil de software.

Os conceitos-chave da Separação Multidimensional de Interesses são:

- Hiperespaço (*hyperspace*): É um espaço multidimensional no qual residem as unidades e que permite a definição de quaisquer dimensões de interesses em qualquer estágio do ciclo de vida. Permite também identificar o relacionamento e a integração dos interesses;
- Unidades: Módulos que possibilitam o encapsulamento de responsabilidades que juntas podem formar um interesse. São classificadas em atômicas e compostas. Unidades atômicas são métodos e atributos enquanto que as compostas são classes, pacotes, componentes, diagramas de colaboração, etc;
- Hiperfatias (*hyperslices*): conjunto de unidades em uma determinada dimensão que determinam um interesse.

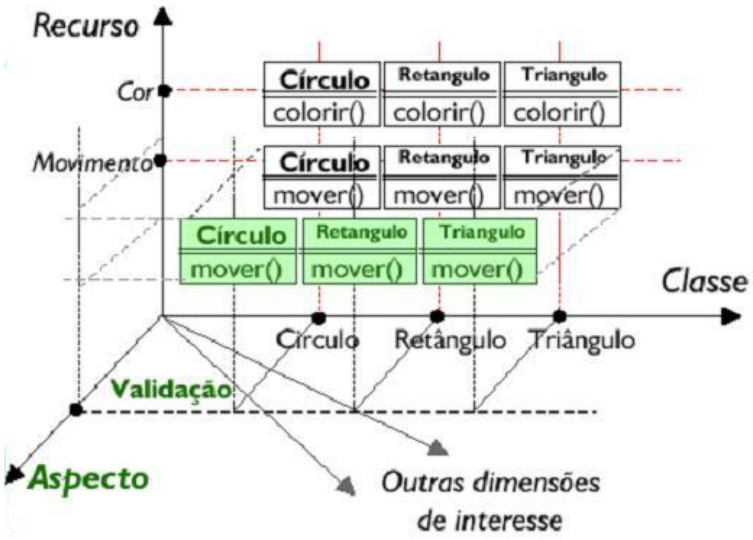


Figura 2.6: Um exemplo de dimensões de interesse.

Como mostra a Figura 2.6, as unidades de um hiperespaço são organizadas em uma matriz multidimensional em que cada eixo representa uma dimensão de interesse e cada ponto no eixo é

um interesse naquela dimensão. Três dimensões são representadas no gráfico: Classe, Recurso e Aspecto. A dimensão *aspecto* possui o interesse *validação* que afeta os métodos *mover* das três classes existentes no sistema: Círculo, Retângulo e Triângulo. Sendo assim, a dimensão *aspecto* determina uma hiperfatia do gráfico que agrupa o comportamento de validação para o sistema. Essa hiperfatia contém as mesmas classes que as outras hiperfatias, porém apenas com o comportamento relativo à validação.

Um processo de desenvolvimento de software fiel ao princípio de Separação de Interesses para a modularização de software deve recomendar que se separem inicialmente os interesses estruturais do sistema e que posteriormente seja realizada a composição deles a fim de obter o sistema completo. Em *Hyper/J*, essa composição consiste em definir pontos de junção correspondentes nas hiperfatias a serem compostas e combiná-las nesses pontos. Exemplos de pontos de junção são: atributos, métodos e classes. A correspondência e os detalhes da composição são especificados por estratégias de composição, fornecidas pelo desenvolvedor, que podem ser divididas em duas categorias: geral e específica. As estratégias gerais são utilizadas para identificar unidades correspondentes nas hiperfatias. As estratégias específicas são utilizadas para definir exceções ou especializações quando as estratégias gerais não se aplicam para todos os casos. A Tabela 2.1 apresenta as regras de composição que são fornecidas pelo *Hyper/J* e uma breve descrição de cada uma.

Quando se usa *Hyper/J*, um desenvolvedor fornece três entradas (Lai et al., 2000):

- um arquivo de hiperespaço que descreve os arquivos de classe Java que podem ser manipulados pelo *Hyper/J*;
- um arquivo de mapeamento de interesses que descreve quais fragmentos de código Java mapeiam-se para cada dimensão de interesse;
- um arquivo de hipermódulo que descreve quais dimensões de interesses (hiperfatias) devem ser integradas e como tal integração deve ser feita.

Na figura 2.7 é mostrado um exemplo de código que pode servir de entrada para o *Hyper/J*. Em 2.7(a) aparece uma classe A, codificada em Java, que representa a primeira versão de um sistema, cujo objetivo é imprimir no console a *string* “Hello”. Eventualmente, o sistema precisa ser evoluído, provavelmente por solicitação de um usuário. Deseja-se que a *string* impressa seja “Hello, world!”. Em Orientação a Objetos, um desenvolvedor iria alterar invasivamente o código-fonte da classe A. Utilizando *Hyper/J*, contudo, é possível para esse caso adicionar o código de evolução em outro módulo, no caso uma classe B, conforme mostrado em 2.7(b). Para compor os dois módulos a fim de formar um sistema completo, utilizam-se as informações contidas nos arquivos em 2.7(c), (d) e (e). A regra de composição *mergeByName* junta as classes A e B, enquanto a regra de composição *equate* especifica que os métodos de mesmo nome serão considerados o mesmo método. Assim, a semântica do programa gerado será equivalente à situação em que o

Regra de Composição	Descrição
mergeByName	Unidades com o mesmo nome, em hiperfatias diferentes, são correspondentes e devem ser combinadas por meio de um relacionamento de intercalação.
nonCorrespondingMerge	Unidades com o mesmo nome e em hiperfatias diferentes não são correspondentes.
overrideByName	Unidades com o mesmo nome, em hiperfatias diferentes, são correspondentes e devem ser combinadas por um relacionamento de sobreposição, em que a última unidade indicada se sobrepõe às anteriores.
equate	As unidades listadas devem corresponder umas às outras, mesmo com nomes diferentes.
order	A ordem de listagem das rotinas será a ordem de execução das rotinas combinadas.
rename	Renomeia a unidade listada como desejado.
merge	Um relacionamento de intercalação entre as unidades listadas, homônimas ou não, deve ocorrer. Equivalem a usar *equate* e *mergeByName* ao mesmo tempo.
noMerge	Faz com que algumas unidades correspondentes não sejam intercaladas, até mesmo se a estratégia geral for *mergeByName* ou *overrideByName*.
override	Uma unidade listada deve sobrepor às demais unidades listadas.
match	Uma dada unidade deve corresponder a um conjunto de unidades que são descritas por padrões de correspondência ou nomes de unidades. Esse relacionamento apenas indica correspondência.
bracket	Indica a ordem de execução de um conjunto de rotinas por meio das palavras-chave *before* e *after*.
set summary function	Cria uma rotina que retorna um array de valores que agrupa o retorno das rotinas originais.

Tabela 2.1: Regras de Composição e Tipos de Relacionamentos do *Hyper/J* (Tarr e Ossher, 2000)

código presente em B.print() seja adicionado após o código de A.print(), permitindo que o sistema seja evoluído de acordo com as necessidades impostas.

A linguagem de composição baseada em texto do *Concern Manipulation Environment* (CME) (The CME Team, 2005) é um descendente direto do *Hyper/J*. A linguagem do CME possui uma sintaxe diferente da sintaxe do *Hyper/J*, mas as capacidades de ambas são fundamentalmente as mesmas, exceto pelo fato de que o CME também trata de interesses estruturais entrecortantes.

O módulo hiperfatia do *Hyper/J* passa a ser denominado interesse (*concern*) no CME. Isso causa um problema de nomenclatura porque o termo "interesse" passa a se referir tanto à noção tradicional, de mais alto nível, dada a este termo, quanto ao módulo que contém a implementação referente a um interesse estrutural.

O CME possui um operador separado para compor um interesse-base com outro interesse-base (**merge concerns**), e um outro operador para compor um interesse-base com um interesse

```
package aTest;
class A {
   void print() {
      System.out.println("Hello");
   }

   static void main(String[] args){
      A a = new A();
      a.print();
   }
}
```

(a) Um exemplo de classe.

```
package aTest;
class B {
   void print() {
      System.out.println(", world!");
   }
}
```

(b) Outro exemplo de classe.

```
hyperspace Test
   composable class aTest.*;
```

(c) Arquivo de Hiperespaço.

```
hypermodule Test
   hyperslices:
      Feature.Kernel;
      Feature.New;
   relationships:
      mergeByName;
      equate class Feature.Kernel.A,
       Feature.New.B;
```

(d) Arquivo de Hipermódulo.

```
class A: Feature.Kernel
class B: Feature.New
```

(e) Arquivo de Mapeamento de Interesses.

Figura 2.7: Exemplo de código de entrada para o Hyper/J (adaptado de (Lai et al., 2000))

estrutural entrecortante (**extend concern with**). Portanto, a linguagem textual do CME distingue explicitamente interesses entre base e entrecortantes. Isso pode causar problemas quando se usa acidentalmente o operador errado para tratar interesses do outro tipo, por exemplo usar o *merge concerns* para compor um interesse-base com um interesse estrutural entrecortante.

Infelizmente, a linguagem de composição baseada em texto do CME está pouco documentada até agora, mas este problema possivelmente será resolvido no futuro próximo. Uma das poucas descrições publicamente disponíveis está presente na mesma fonte que descreve a abordagem *Theme/AOSD* de modelagem de *design* (comentada no próximo capítulo) (Clarke e Baniassad, 2005).

2.6 Considerações Finais

A discussão presente neste capítulo fornece embasamento para se criar uma abordagem que una o melhor das abordagens DSOA com *AspectJ* e *MDSoC/Hyperspaces*, a fim de superar as deficiências existentes no POO. No próximo capítulo será mostrado como os conceitos de espalhamento, entrelaçamento, interesses estruturais e tirania da decomposição dominante se relacionam para formar uma nova abordagem.

CAPÍTULO

3

Características do Desenvolvimento de Software Orientado a Temas

3.1 Considerações Iniciais

No Capítulo 2 foi apresentado o contexto no qual este trabalho está inserido, e também algumas deficiências do POO foram sugeridas. Neste capítulo, oferece-se a motivação para o Desenvolvimento de Software Orientado a Temas (DSOT) e como funciona a semântica de dois mecanismos importantes nesta abordagem, a composição e a redefinição.

Alguns exemplos deste capítulo estão ilustrados por meio de *diagramas de temas*, estabelecidos em uma modificação da notação *Theme/UML*. A notação *Theme/UML* original é parte de uma abordagem de análise e modelagem de *design* de software denominada Abordagem *Theme* para o Desenvolvimento de Software Orientado a Aspectos (aqui abreviado como Theme/DSOA) (Clarke e Baniassad, 2005).

Para a análise, *Theme/DSOA* define uma notação denominada *Theme/Doc*, enquanto para a modelagem de *design* é definida a notação *Theme/UML*, uma extensão da UML padrão própria para DSOA. Um dos objetivos de *Theme/AOSD* é ser o mais independente possível de abordagens de implementação específicas, como *AspectJ* e *Hyper/J* (discutidas no Capítulo 2), para que os modelos de *design* possam ser implementados em diferentes tecnologias de maneira equivalente.

A notação *Theme/UML* original foi alterada para melhor se adequar ao propósito deste trabalho, que é demonstrar as idéias presentes no DSOT. As alterações sobre a notação original são apresentadas em inglês no Anexo A.

Este capítulo se divide da seguinte forma. Na Seção 3.2 é mostrada informalmente a filosofia original da abordagem de DSOT. Em seguida, as Seções 3.3 e 3.4 definem mais rigorosamente a semântica dos mecanismos de composição e de redefinição do DSOT, respectivamente.

3.2 Motivação para o Desenvolvimento de Software Orientado a Temas

Como demonstrado no Capítulo 2, espalhamento e entrelaçamento são problemas inevitáveis de evoluibilidade no POO pois não é possível modularizar a implementação relativa a interesses estruturais. Isso ocorre por causa da tirania da decomposição dominante: o POO permite visualizar software somente sob a *dimensão de tipos de dados* (modularização por classes), restringindo qualquer outra forma de visualização e manipulação do software. Apesar destes problemas, a dimensão de tipos de dados tem se provado muito útil para desenvolver software: a funcionalidade completa do sistema de software resulta do comportamento que emerge da interação de todos os interesses estruturais relevantes, e este comportamento emergente é o que se obtém visualizando software sob a dimensão de tipos de dados. Até mesmo com as limitações discutidas anteriormente, não se quer perder a capacidade de visualizar software sob esta dimensão.

Entretanto, no tempo de desenvolvimento seria útil aos desenvolvedores manipular também a implementação de cada interesse estrutural separadamente. Em outras palavras, seria conveniente durante o desenvolvimento também poder visualizar software sob a *dimensão de interesses estruturais*. Esta dimensão é complementar à tradicional dimensão de tipos de dados: tais dimensões exprimem perspectivas diferentes sobre um mesmo programa e, quando a implementação em uma das dimensões for alterada, as alterações feitas serão imediatamente refletidas na outra dimensão. A possibilidade de alternar a visualização e a manipulação de software entre as duas dimensões fornece grande poder aos desenvolvedores.

Um programa completo pode ser encarado de duas maneiras: sob a dimensão de interesses estruturais, ele é o resultado da interação entre todos os interesses estruturais requeridos; sob a dimensão de tipos de dados, o programa completo é uma estrutura de classes tradicional como no POO.

Durante o processo de *design*, pode-se visualizar a estrutura de classes universal do programa completo, que contém a implementação de todos os interesses estruturais necessários para formar o programa. Deve ser possível também, pela dimensão de tipos de dados, visualizar uma implementação parcial da estrutura de classes universal que lida com elementos referentes a somente um interesse estrutural, ignorando o restante da implementação, que se refere a outros interesses estruturais. Na dimensão de interesses estruturais, essa implementação parcial da estrutura de classes é encarada como um módulo, denominado *tema* (Clarke e Baniassad, 2005). Assim, ao se decompor o programa por elementos correspondentes a interesses estruturais, cada uma dessas partes do programa irá corresponder a um tema.

A operação inversa também pode ser feita. Se forem desenvolvidas, sob a dimensão de tipos de dados, várias implementações parciais de uma estrutura de classes, e cada uma delas corresponder a um interesse estrutural relevante para o programa completo, pode-se obter o programa completo (isto é, a estrutura de classes completa) por meio da combinação de todas as implementações parciais da estrutura de classes. Na dimensão de interesses estruturais, a operação equivalente é desenvolver vários temas primitivos, em que cada um deles corresponde a um interesse estrutural, e em seguida efetuar a composição deles para se obter o programa completo (isto é, o tema final). Esta discussão considera que a composição entre dois ou mais temas resulta em um novo tema.

Como, sob a dimensão de tipos de dados, o programa completo é uma estrutura de classes tradicional, conclui-se que sobre o DSOT o programa conserva todas as propriedades que ele teria se fosse implementado no POO puro desde o início. Sob a dimensão de tipos de dados, são tratadas classes, herança, polimorfismo, e todos os conceitos importantes do POO. Sob a dimensão de interesses estruturais, preocupa-se com temas e composições. Por meio do DSOT, pode-se alternar entre estas dimensões quando necessário. É precisamente essa possibilidade de alternar suavemente entre as duas dimensões que traz todo o poder da abordagem de DSOT. Portanto, o DSOT fornece novas habilidades ao desenvolvedor acostumado a utilizar POO, sem suprimir nenhuma das propriedades originais que um programa orientado a objetos teria.

Uma pilha frequentemente não corresponde a uma parte significativa da arquitetura de um programa típico, mas é bastante utilizada como um exemplo canônico que ilustra a motivação para a teoria de tipos abstratos de dados e, por extensão, do POO. Analogamente, partes de implementação correspondentes ao *logging* da execução de algumas rotinas de um programa frequentemente não correspondem a uma parte significativa da arquitetura de um programa típico, mas são bastante utilizadas como um exemplo canônico que ilustra a motivação para a teoria de dimensão de visualização de interesses estruturais e, por extensão, do DSOA em geral e DSOT em particular. Assim, a modularização de *logging* é utilizada a seguir como exemplo de como funciona em termos gerais a modularização por meio de temas.

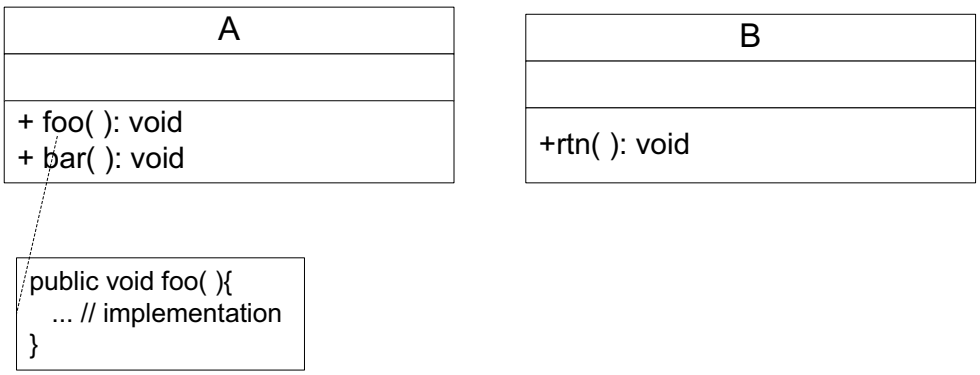


Figura 3.1: A rotina *foo( )* da classe *A* deve ser sujeita ao *logging*.

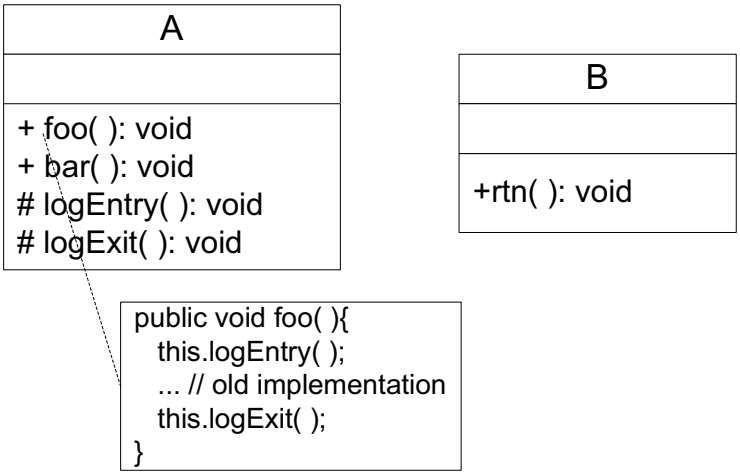


Figura 3.2: Rotina *foo( )* após ser alterada invasivamente para adicionar *logging*.

Suponha uma classe *A* com uma rotina *foo( )* que deve estar sujeita a uma forma de *logging* (Figura 3.1). No caso de POO, para inserir o código correspondente ao *logging*, seria necessário efetuar uma mudança invasiva, como mostrado na Figura 3.2. Mas em DSOT não é necessário fazer uma mudança invasiva. O segredo é manipular o software sob a dimensão de interesses estruturais. Considera-se que a implementação original da classe *A* está presente em um tema, chamado aqui de **CoreFunctionality**. Então, cria-se um novo tema denominado **Logging** para guardar a implementação da funcionalidade de *logging* independentemente das demais funcionalidades do programa (Figura 3.3).

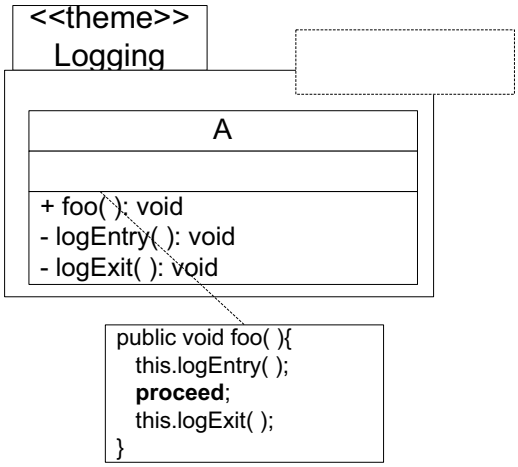


Figura 3.3: Tema **Logging**, implementado para adicionar *logging* à rotina *foo( )* da classe *A*.

Note que a rotina *foo( )* de **Logging** é *incompleta*, um conceito que não existia no POO. Isso significa que sua implementação deve ser completada por implementação presente em provavelmente outro tema. A palavra-chave *proceed* no corpo da rotina indica que a rotina é incompleta, e que no ponto em que *proceed* aparece deve ser adicionada a implementação que completa a rotina. Assim, a parte substituível de uma rotina incompleta consiste no marcador de lugar (*placeholder*) *proceed* para o código de outra rotina (provavelmente presente em outro tema).

Para formar a implementação completa, basta compor na dimensão de interesses estruturais o tema **CoreFunctionality** com o tema **Logging**. O resultado da composição, visualizado sob a dimensão de tipos de dados, é exatamente o que aparece na Figura 3.2.

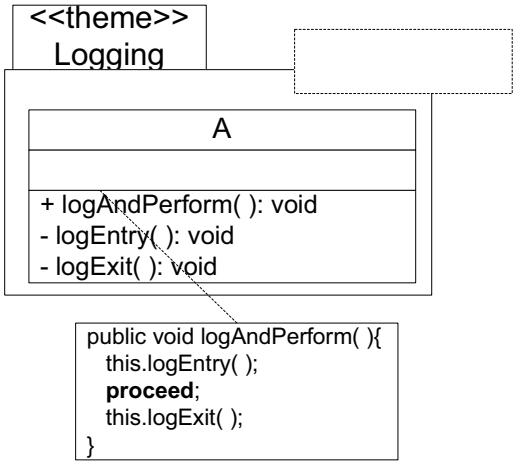


Figura 3.4: Tema **Logging**, implementado para adicionar *logging* às rotinas *foo( )* e *bar( )* da classe *A*.

Suponha agora que a rotina *bar( )* da classe *A* também tenha de ser afetada pelo *logging*. Neste caso, devemos tornar a implementação do tema **Logging** mais genérica por meio de uma refatoração simples, para que a rotina de *logging* ainda pertença à classe *A* mas não seja correspondente somente à rotina *foo( )* de **CoreFunctionality**: a rotina *foo( )* deve ser renomeada, por exemplo para *logAndPerform( )* (Figura 3.4). Para compor **CoreFunctionality** com **Logging**, deve-se indicar que *logAndPerform( )* corresponde com as rotinas *foo( )* e *bar( )*. Isso é feito por meio de uma *redefinição*. A rotina *logAndPerform( )* do tema **Logging** é redefinida, gerando um tema anônimo (Figura 3.5). Esse tema é composto com **CoreFunctionality**, gerando o programa desejado (Figura 3.6).

Agora suponha que uma nova evolução no programa é necessária: a rotina *rtn( )* da classe *B* também tem de ser afetada pelo *logging*. Então, refatora-se novamente a implementação do tema **Logging** para torná-la ainda mais genérica: a implementação de *logAndPerform( )* não pertence mais somente à classe *A*. Nesse caso, a classe *A* do tema **Logging** é renomeada para *Logable* (Figura 3.7). Outra redefinição é feita, fazendo com que a classe *Logable* do tema **Logging** corresponda às classes *A* e *B* do tema **CoreFunctionality**, e que a rotina *logAndPerform( )* passe a corresponder também à rotina *rtn( )* da classe *B* do tema **CoreFunctionality**. O tema anônimo resultante

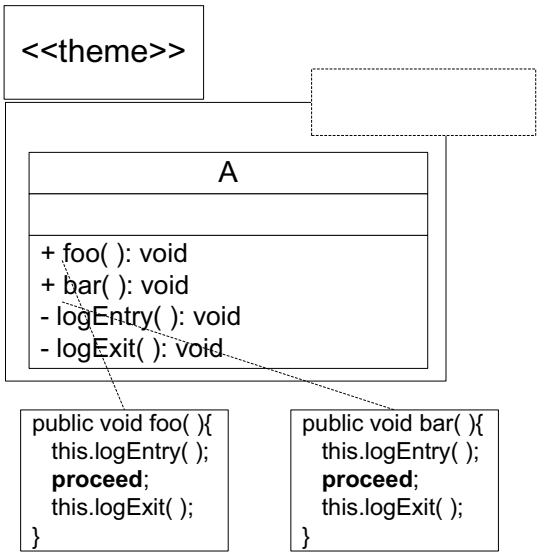


Figura 3.5: Tema anônimo que resulta da redefinição de *logAndPerform( )* para *foo( )* e *bar( )*.

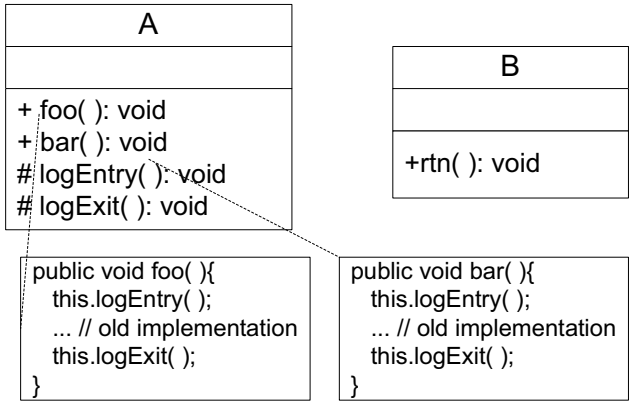


Figura 3.6: *Logging* sobre as rotinas *foo( )* e *bar( )* da classe *A*.

da redefinição de **Logging** (Figura 3.8) é então composto com **CoreFunctionality**, gerando o programa final.

Note que as refatorações que foram feitas tiveram como objetivo tornar a implementação do tema **Logging** mais geral, necessidade que não havia sido antecipada. Se a implementação tema **Logging** genérico tivesse sido usada desde o início, quando se desejava apenas efetuar o *log* da execução da rotina *foo( )* da classe *A*, nenhuma refatoração teria sido feita. Na implementação do POO, ao contrário, as refatorações feitas não funcionam para as versões antigas do programa, tendo que ser desfeitas para que os módulos se tornem compatíveis com as versões antigas.

O exemplo demonstra que há três mecanismos importantes envolvidos na abordagem de DSOT: o *tema*, módulo que contém a implementação relativa a um interesse estrutural na dimensão de interesses estruturais; a *composição*, operação que recebe dois temas como argumentos e retorna um tema que contém a implementação composta dos dois argumentos; e a *redefinição*, operação

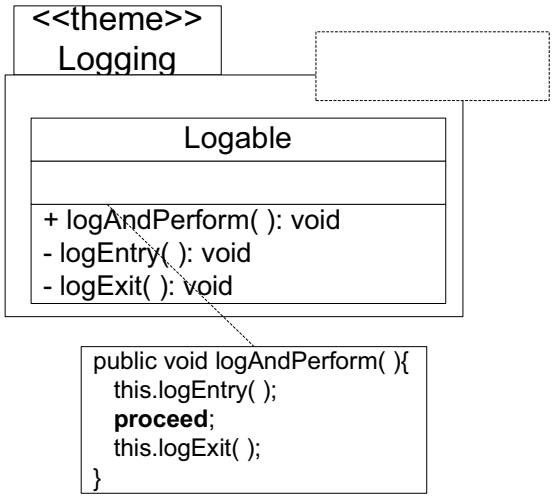


Figura 3.7: Tema **Logging**, implementado para adicionar *logging* às rotinas *foo( )* e *bar( )* da classe *A*, e à rotina *rtn( )* da classe *B*.

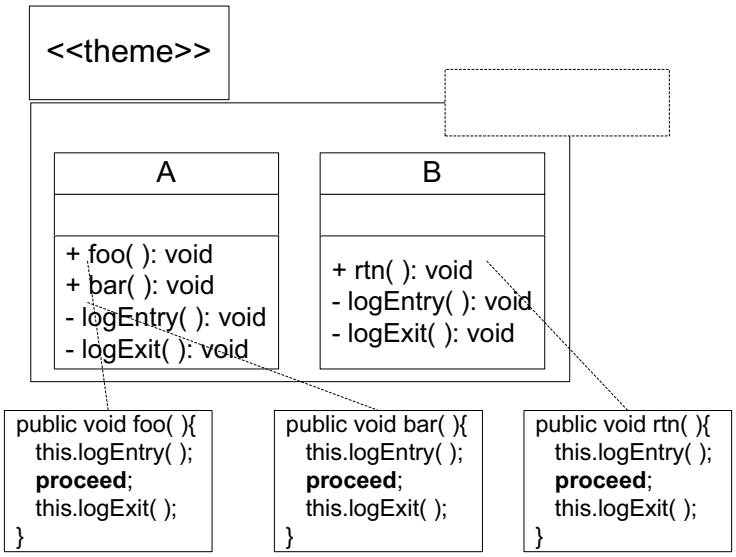


Figura 3.8: Tema anônimo que resulta da redefinição de *logAndPerform( )* para *foo( )* e *bar( )* da classe *A*, e para *rtn( )* da classe *B*.

que recebe um tema e as assinaturas de alguns módulos internos ao tema como parâmetros e retorna um tema que contém a implementação redefinida do tema original. Os operadores de composição e de redefinição possuem a propriedade de *fechamento*: o resultado das operações sempre é um tema válido. Isso implica que o resultado de uma fórmula (expressão) de composição pode ser utilizado

em outra fórmula de composição como se fosse um tema primitivo, possibilitando assim a criação de expressões aninhadas. A composição e a redefinição podem aparecer de forma combinada em uma fórmula de composição.

Ao fim de um projeto bem sucedido que utiliza a tecnologia de DSOT, o seguinte cenário deve estar presente:

- Os requisitos foram definidos e modelos de análise foram criados;
- Temas foram identificados e *relacionamentos de composição* entre temas foram determinados;
- Cada *tema primitivo* foi internamente modelado e implementado como um "programa" orientado a objetos *declarativamente completo* (explicado a seguir);
- Os relacionamentos de composição foram refinados para tratar exceções (módulos internos a temas que tiveram de ser redefinidos);
- Esses relacionamentos foram implementados por meio da criação de uma *fórmulas de composição*;
- O programa completo foi composto passo a passo por uma *ferramenta de composição*.

Isto não significa que o processo de desenvolvimento / evolução deve obedecer a ordem dos itens expostos; significa apenas que após o projeto todas as atividades mencionadas devem ter sido efetuadas de alguma forma. Isto é, observada após o projeto, a solução traz a ilusão de ter sido desenvolvida por meio de uma sequência perfeita de atividades, o que não corresponde à realidade do projeto, mas permite uma explicação didática de como a solução funciona e dos compromissos envolvidos no *design*.

Os processos de definição de requisitos e de análise do problema estão fora do escopo desta dissertação. Apenas se assume aqui que os requisitos foram definidos de forma tal que a identificação de temas importantes é razoavelmente direta.

Um requisito especial deve ser atendido para que um tema seja considerado válido: o tema deve ser *declarativamente completo*. Este conceito é tão importante para a abordagem de DSOT que a ele é dedicada uma subseção à parte.

3.2.1 Completeza Declarativa

Um tema não precisa conter a implementação de todas as rotinas que ele usa; somente as rotinas que são "essenciais" a um interesse estrutural são implementadas no tema que corresponde àquele interesse estrutural. A única restrição a ser observada é que um tema deve declarar todas as rotinas que ele usa, até mesmo as que não são implementadas por aquele tema. Declarar uma rotina não-implementada significa implementá-la como rotina abstrata, e como consequência, implementar a

classe que a contém como uma classe abstrata (de implementação incompleta). Esta restrição a ser satisfeita é denominada *completeza declarativa*.

No POO, uma classe abstrata pode conter somente rotinas abstratas e completas, pois o conceito de rotinas incompletas não existe naquela abordagem. Em DSOT, uma classe abstrata pode conter também rotinas incompletas. Uma classe não-abstrata de DSOT pode conter somente rotinas completas, assim como no POO. Isto significa que, se uma classe contém ao menos uma rotina incompleta, então tal classe é abstrata, o que significa que ela terá de ser complementada por herança ou por composição com código extra para que possa ser instanciada em tempo de execução.

No POO, a regra de não poder instanciar classes abstratas tem sentido, já que uma classe abstrata não poderia tornar-se completa; somente suas subclasses poderiam fornecer a implementação das rotinas que não foram implementadas na classe abstrata, portanto, se necessário, a subclasse é instanciada. Diferentemente da orientação a objetos, em DSOT um tema pode conter código que indica a instanciação de uma classe abstrata declarada naquele tema. Em tempo de compilação isto será válido. Isto é útil porque a classe abstrata em um tema pode conter somente o que é essencial para aquele tema. Detalhes extras de implementação podem ser fornecidos em outro tema. Antes que o programa seja executado, contudo, aquela classe deverá se tornar completa, por exemplo por meio da composição daquele tema com outro tema que complete a classe abstrata.

No POO, a criação de uma rotina ao mesmo tempo abstrata e privada em uma classe não tem sentido. Mas no DSOT isso é válido, e indica que a rotina deve ser composta com uma rotina não-abstrata, que pode ser privada ou até mesmo pública.

Dessa forma, graças à restrição de completeza declarativa, software criado por meio de DSOT possui duas diferenças importantes com relação a software orientado a objetos tradicional:

- É possível instanciar objetos de classes abstratas.
- Podem existir rotinas que são incompletas (podem conter uma instrução **proceed**. Na modificação da notação *Theme/UML* que aparece no Anexo A, elas são marcadas com o estereótipo «incomplete». Assim, um tema pode conter classes cujas rotinas são abstratas (não-implementadas), incompletas (parcialmente implementadas) ou completas (totalmente implementadas). Rotinas abstratas e incompletas podem aparecer somente em classes abstratas.

O Capítulo 4 descreve um estudo de caso no qual aparecem várias rotinas criadas para satisfazer a restrição de completeza declarativa de temas.

3.2.2 História da abordagem de Desenvolvimento de Software Orientado a Temas

O DSOT é uma abordagem que surgiu a partir de duas atividades efetuadas de forma intercalada:

- Observação das vantagens, deficiências e omissões teóricas e práticas das abordagens de implementação *Hyper/J* e *AspectJ*, e da abordagem de modelagem de *design Theme/DSOA* (Clarke e Baniassad, 2005);
- desenvolvimento de um estudo de caso que consistiu na implementação de um programa para
  - experimentar e refinar os mecanismos que se acreditavam necessários para a abordagem e
  - demonstrar evidências de que o uso de tais mecanismos é vantajoso.

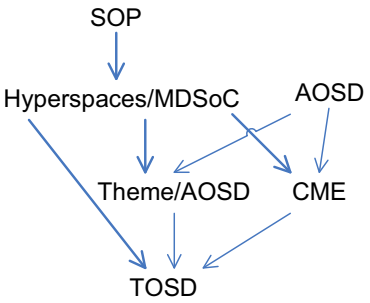


Figura 3.9: Histórico de evolução da abordagem de DSOT. As setas mais espessas representam maior influência. As siglas das abordagens estão em inglês.

A maior parte da fundamentação teórica do DSOT é resultado da evolução da teoria presente na abordagem *MDSoC/Hyperspaces* (Tarr et al., 1999; Ossher e Tarr, 2001; Tarr e Ossher, 2000; Ossher e Tarr, 1999; IBM Research, 2000), a qual é é em si mesmo uma evolução da abordagem *Subject-Oriented Programming* (SOP) (Harrison e Ossher, 1993). O nome "tema" para o módulo de primeira ordem no DSOT foi adotado com base na abordagem *Theme/DSOA* (Clarke e Baniassad, 2005). A instrução *proceed* e o conceito de rotinas incompletas foram inspirados nos adendos do tipo *around( )* de *AspectJ* (The AspectJ Team, 2006). A influência de cada abordagem anterior na definição de DSOT é mostrada na Figura 3.9.

Nesta seção foi dada uma visão informal sobre a semântica da composição e da redefinição com base em um exemplo simples. As subseções a seguir detalham com maior rigor a semântica de composição e de redefinição para cada um dos cenários que apareceram ao longo desta pesquisa. De forma alguma afirma-se que os cenários listados a seguir são todos os cenários possíveis que podem aparecer em um programa; no estudo de caso descrito no Capítulo 4, contudo, somente os cenários descritos aqui ocorreram na prática.

3.3 Semântica da Composição de Temas

Na abordagem de implementação definida aqui para o DSOT, o operador de composição é o +, que recebe dois temas como argumento. Há vários cenários nos quais o operador + está envolvido, e cada um dos cenários produz um resultado diferente para a composição.

Quando ocorre uma composição entre dois temas, uma classe de um dos temas pode *corresponder* a uma classe de outro tema. A correspondência indica que as duas classes são de certa forma a mesma classe, tratada em perspectivas diferentes por cada um dos temas, que encapsulam implementação correspondente a interesses estruturais diferentes. Outra classe de um tema pode também não corresponder a nenhuma classe do outro tema.

Por assumir que o desenvolvedor procura tornar o programa mais claro, considera-se que classes correspondentes em uma composição são somente as classes que possuem o mesmo nome. Atributos correspondentes, para o DSOT, são atributos que estão em classes correspondentes e possuem o mesmo nome. Rotinas correspondentes são rotinas que estão em classes correspondentes e possuem a mesma assinatura (nome da rotina e natureza dos parâmetros, que inclui quantidade, ordem e tipo). Assim, a abordagem de DSOT procura forçar o desenvolvedor a tornar o programa mais compreensível, utilizando estratégia compatível com a cognição humana de dar o mesmo nome para coisas iguais e dar nomes diferentes para coisas distintas.

Uma composição sempre ocorre entre dois temas. Uma aparente composição de mais de dois temas é, na verdade, uma sequência de composições, na qual o resultado da composição entre os dois primeiros temas é composto com o terceiro tema, resultando em um tema que deve ser composto com o quarto tema, e assim por diante, até resultar no tema final daquela cadeia de composições. É importante mencionar que o operador + não possui nem a propriedade comutativa nem a associativa, pois a ordem dos temas listados pode influenciar o resultado final da composição. A influência da ordem dos temas é melhor explicada na Subseção 3.3.4.

A composição demonstrada em todos os cenários a seguir é implementada conforme se segue.

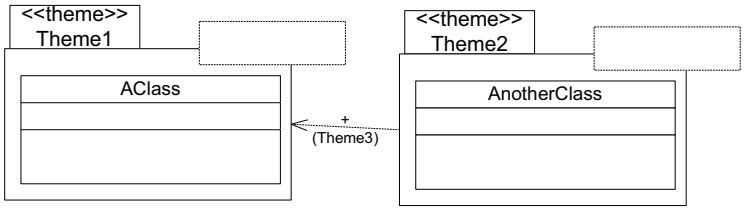
```
Theme3 := Theme1 + Theme2;
```

3.3.1 Classes não-correspondentes

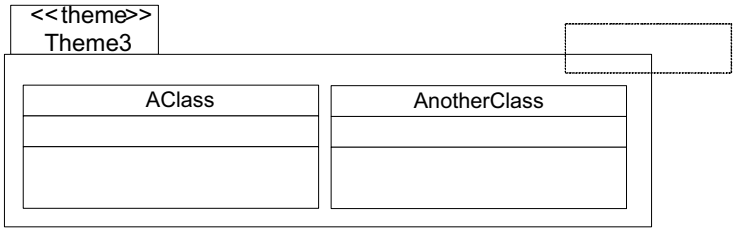
Se o desenvolvedor desejar que dois temas sejam compostos, e um dos temas possui uma classe que o outro tema não possui, é razoável assumir que esta classe deve aparecer no tema resultante. Assim, se uma classe de um tema não corresponder a nenhuma classe do outro tema, a classe é copiada de forma intacta para o tema resultante. A semântica de composição para este cenário é demonstrada na Figura 3.10.

3.3.2 Classes correspondentes, atributos e/ou rotinas não-correspondentes

Se o desenvolvedor desejar que duas classes sejam combinadas, e uma das classes possui elementos internos (rotinas ou atributos) que a outra classe não possui, é razoável assumir que estes elementos devem aparecer no tema resultante. Assim, se uma classe de um tema corresponder a uma classe de outro tema, e houver atributos e/ou rotinas que não possuem correspondentes,



(a) Diagrama de temas.



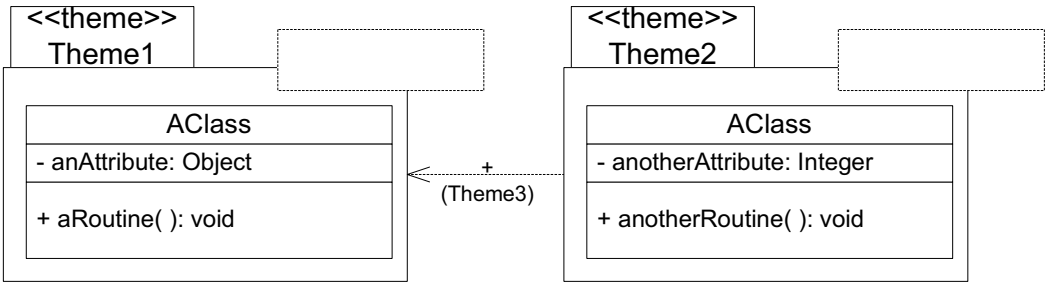
(b) Diagrama de classes do tema resultante.

Figura 3.10: Exemplo de composição no qual uma classe não possui correspondência no outro tema.

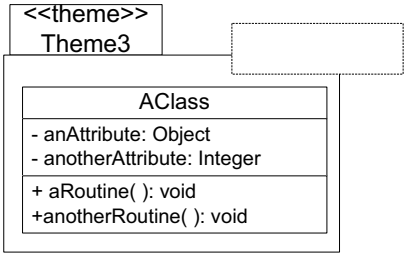
estes atributos e/ou rotinas são copiados de forma intacta para o tema resultante. A semântica de composição para este cenário é demonstrada na Figura 3.11.

3.3.3 Classes correspondentes, atributos correspondentes

Se o desenvolvedor desejar que dois atributos sejam combinados, é razoável assumir que o desenvolvedor considera ambos o mesmo atributo. Assim, se uma classe de um tema corresponder a uma classe de outro tema, e houver um atributo que possui correspondente, este atributo é copiado de forma intacta somente uma vez para o tema resultante. A semântica de composição para este cenário é demonstrada na Figura 3.12.

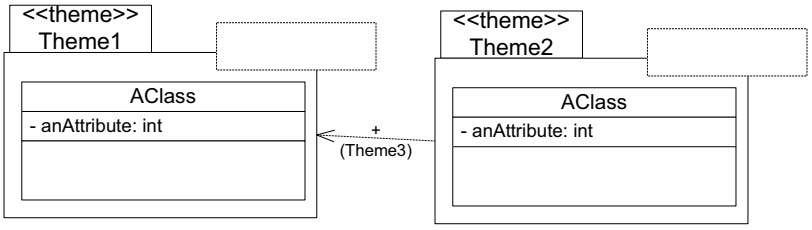


(a) Diagrama de temas.

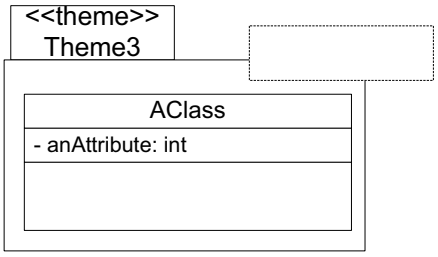


(b) Diagrama de classes do tema resultante.

Figura 3.11: Exemplo de composição no qual um atributo e uma rotina não possuem correspondência no outro tema.



(a) Diagrama de temas.



(b) Diagrama de classes do tema resultante.

Figura 3.12: Exemplo de composição no qual um atributo possui correspondência no outro tema.

3.3.4 Classes correspondentes, rotinas completas correspondentes

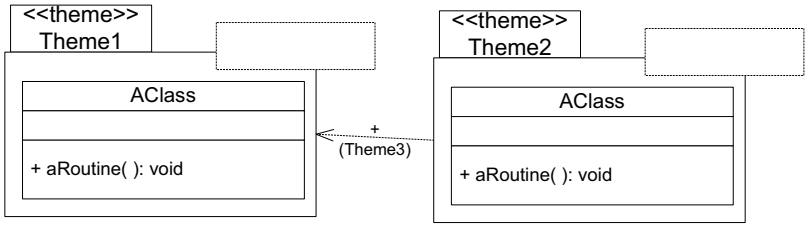
Uma rotina completa contém instruções que efetuam um determinado processamento que interessa para o sistema globalmente, como uma de suas "engrenagens". Se o desenvolvedor desejar que duas rotinas completas sejam combinadas, é razoável assumir que o desenvolvedor considera ambas a mesma rotina. Uma rotina completa sempre possui implementação interna. A rotina de cada tema contém uma implementação específica para aquele tema. Parece então razoável assumir que no tema resultante da composição deve aparecer a implementação específica das rotinas de ambos os temas concatenada em uma rotina só. Assim, se uma classe de um tema corresponder a uma classe de outro tema, e houver uma rotina completa que corresponde a outra rotina completa, as duas rotinas se unem em uma só. A ordem das implementações é a seguinte: aparece primeiro a implementação da rotina do primeiro tema listado na fórmula de composição, e em seguida a implementação da rotina do segundo tema listado. Isso mostra porque o operador de composição não é associativo nem comutativo. Eventuais conflitos de nomes de variáveis locais de rotinas podem ser resolvidos por meio de renomeação automática das variáveis. A semântica de composição para este cenário é demonstrada na Figura 3.13.

3.3.5 Classes correspondentes, rotina abstrata correspondente a outra rotina qualquer

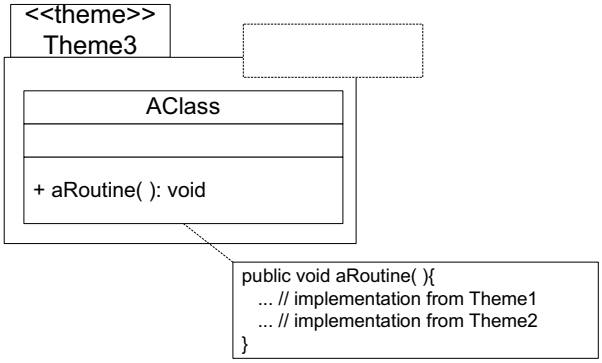
Rotinas abstratas são frequentemente criadas quando em um tema se deseja utilizar uma rotina cuja implementação real estará contida em outra rotina. Rotinas abstratas são fornecidas para satisfazer a restrição de completeza declarativa do tema. Assim, se uma classe de um tema corresponder a uma classe de outro tema, e houver uma rotina abstrata que corresponde a outra rotina (seja ela abstrata, incompleta ou completa), a segunda rotina mencionada é copiada de forma intacta para o tema resultante. A semântica de composição para este cenário é demonstrada na Figura 3.14.

3.3.6 Classes correspondentes, rotina incompleta correspondente a outra rotina não-abstrata

Um desenvolvedor cria uma rotina incompleta quando ele deseja que aquela rotina complemente outra rotina, provavelmente presente em outro tema. Assim, se uma classe de um tema corresponder a uma classe de outro tema, e houver uma rotina incompleta que corresponde a outra rotina não-abstrata (seja ela incompleta ou completa), a primeira rotina mencionada é copiada de forma intacta para o tema resultante, com a exceção de que o marcador **proceed** é substituído pela implementação da segunda rotina mencionada. A semântica de composição para este cenário é demonstrada na Figura 3.15.



(a) Diagrama de temas.

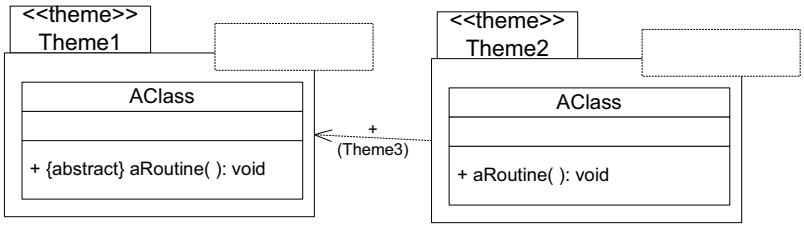


(b) Diagrama de classes do tema resultante.

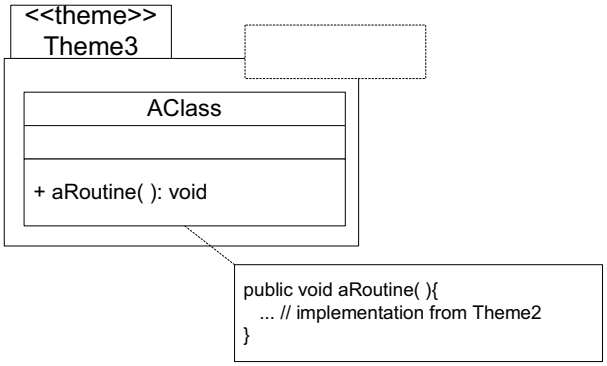
Figura 3.13: Exemplo de composição no qual uma rotina completa possui correspondência no outro tema.

3.3.7 Classes correspondentes, uma classe herda de C1 enquanto a outra classe herda de C2 que é subclasse de C1

Um desenvolvedor tipicamente implementa um tema ignorando temporariamente outros temas relacionados, para que o módulo atinja inconsciência (*obliviousness*) (Filman e Friedman, 2000). Em alguns casos, uma classe de um tema herda de uma determinada classe C1 que é suficiente para seus propósitos. Em outro tema, a mesma classe herda de uma subclasse C2 da antiga superclasse C1 porque precisa de algumas características que a subclasse C2 fornece mas que a superclasse C2 não fornece. Quando os dois temas são compostos, é razoável assumir que o desenvolvedor deseje que a classe implementada herde da superclasse mais específica (C2). Assim, se uma classe de um tema corresponder a uma classe de outro tema, e uma das classes herdar de uma classe C1 enquanto a outra classe herda de uma classe C2 que, por sua vez, é subclasse de C1, a classe resultante herda de C2. A semântica de composição para este cenário é demonstrada na Figura 3.16.



(a) Diagrama de temas.



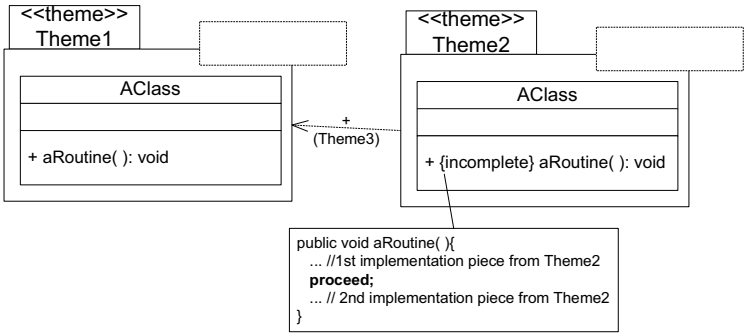
(b) Diagrama de classes do tema resultante.

Figura 3.14: Exemplo de composição no qual uma rotina abstrata possui correspondência no outro tema.

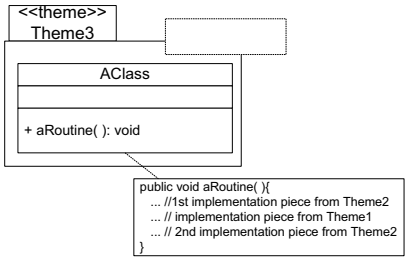
3.4 Semântica da Redefinição de Elementos Internos a Temas

Em alguns casos, antes de efetuar uma composição entre dois temas deseja-se tornar correspondentes alguns elementos que não são correspondentes por *default* conforme a semântica de composição. Os conflitos de correspondência mais comuns são elementos (classes, rotinas ou atributos) com o mesmo nome / assinatura em diferentes temas mas que não devem ser compostos, e elementos com nomes / assinaturas diferentes mas que devem ser compostos. Em tais casos, é necessário redefinir elementos de um dos temas para que a composição ocorra como desejado.

Na abordagem de implementação definida aqui para o DSOT, o operador de redefinição é o *{}*, que recebe um tema e algumas informações internas a aquele tema como argumentos. O operador *{}* retorna um tema que corresponde exatamente ao tema passado como argumento, exceto pelas redefinições que foram especificadas pelo desenvolvedor.

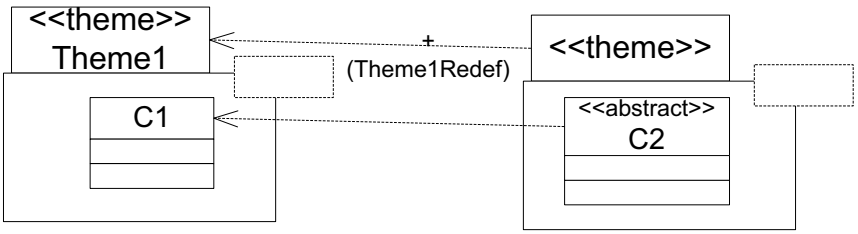


(a) Diagrama de temas.

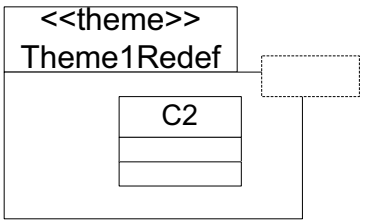


(b) Diagrama de classes do tema resultante.

Figura 3.15: Exemplo de composição no qual rotina incompleta possui correspondência no outro tema.



(a) Diagrama de temas.

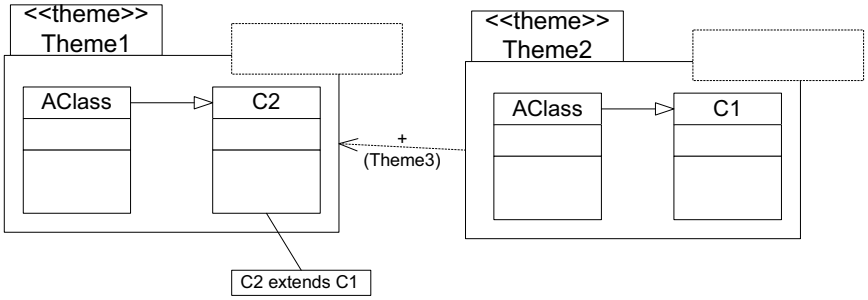


(b) Diagrama de classes do tema resultante.

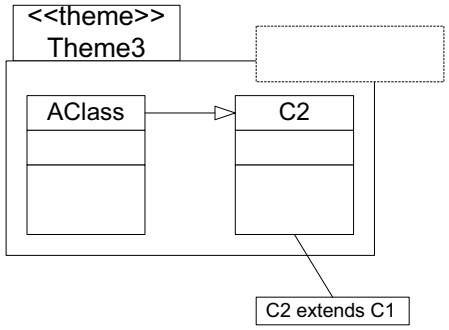
```
Theme1Redef := Theme1{
      C1 => C2;
};
```

(c) Fórmula de composição.

Figura 3.17: Exemplo de redefinição no qual o identificador de uma classe é alterado.



(a) Diagrama de temas.



(b) Diagrama de classes do tema resultante.

Figura 3.16: Exemplo de composição no qual uma superclasse de um tema é subclasse de outra superclasse no outro tema.

Quando um módulo é redefinido, toda a implementação interna àquele tema que utiliza o módulo diretamente é atualizada para utilizar o módulo com a nova definição. Por exemplo, se o identificador de uma classe for alterado, todas as classes que declararem internamente variáveis daquela classe sofrerão atualização do tipo das variáveis.

Em uma especificação de composição, quando algumas redefinições são combinadas com uma composição, o conjunto de redefinições é chamado de *listagem de exceções*, pois elas estabelecem exceções à regra geral de que apenas módulos homônimos internos a temas sujeitos à composição podem se compor em um módulo só.

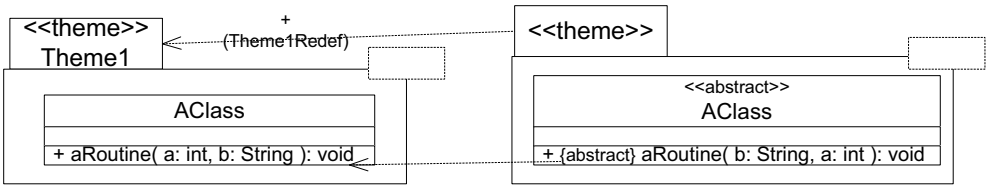
As subseções a seguir mostram como funciona a semântica da redefinição para vários cenários possíveis.

3.4.1 Alteração do identificador de uma classe, de um atributo, de uma rotina ou de um parâmetro de rotina

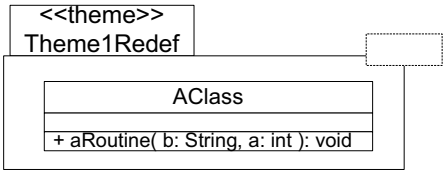
Uma classe, atributo, rotina ou parâmetro pode ter seu identificador alterado por meio de uma redefinição. A semântica de redefinição para este cenário é demonstrada na Figura 3.17.

3.4.2 Alteração da ordem dos parâmetros de uma rotina

A ordem dos parâmetros de rotina pode ser alterada por meio de uma redefinição. A semântica de redefinição para este cenário é demonstrada na Figura 3.18.



(a) Diagrama de temas.



(b) Diagrama de classes do tema resultante.

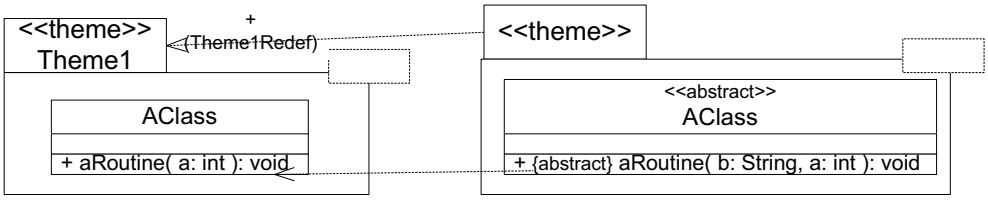
```
Theme1Redef := Theme1{
        AClass.aRoutine( int a, String b ) => aRoutine( String b, int a );
};
```

(c) Fórmula de composição.

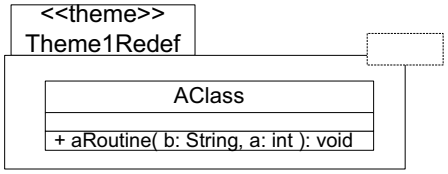
Figura 3.18: Exemplo de redefinição no qual a ordem dos parâmetros de uma rotina é alterada.

3.4.3 Adição de parâmetros a uma rotina

Parâmetros podem ser adicionados a uma rotina por meio de uma redefinição. A restrição neste caso é que, se houver algum código naquele tema que chama a rotina a ser redefinida, tal código deve ser atualizado ao mesmo tempo que a ocorrência da redefinição. No estudo de caso descrito no Capítulo 4, contudo, não apareceram as condições que impõem a necessidade de satisfazer a restrição mencionada. A semântica de redefinição para este cenário é demonstrada na Figura 3.19.



(a) Diagrama de temas.



(b) Diagrama de classes do tema resultante.

```
Theme1Redef := Theme1{
        AClass.aRoutine( int a ) => aRoutine( String b, int a );
};
```

(c) Fórmula de composição.

Figura 3.19: Exemplo de redefinição no qual parâmetros são adicionados a uma rotina.

3.5 Considerações Finais

O estudo de caso que ilustra como os mecanismos fornecidos pelo DSOT são utilizados no contexto de um projeto real é apresentado no Capítulo 4.

CAPÍTULO

4

Estudo de Caso

4.1 Considerações Iniciais

Nesta seção são apresentados o estudo de caso que exemplifica o uso da abordagem DSOT e algumas observações sobre o processo de desenvolvimento utilizado na implementação do exemplo.

O estudo de caso consiste no desenvolvimento de um sistema de edição de textos, chamado *Simple Notepad*, que contempla um subconjunto das características do Bloco de Notas (*Notepad*) do sistema operacional *Windows*. Tal tipo de sistema foi escolhido pelos seguintes motivos: o autor é usuário de sistemas desse tipo, e, portanto, conhece bem o vocabulário do domínio do problema; há boas referências publicadas sobre tal domínio (discutidas na Subseção 4.2.3); e o desenvolvimento de sistemas de edição de textos não é trivial, mas também não é tão complexo a ponto de inviabilizar o projeto de pesquisa.

Para o estudo de caso, a solução final de cada iteração será descrita em detalhes, ao passo que apenas desafios e questões notáveis do processo serão mencionadas. É difícil descrever em detalhes toda a dinâmica de um processo sujeito a criatividade, que envolve várias atividades que se intercalam em um fluxo rápido.

O restante deste capítulo está organizado como se segue. Na Seção 4.2 são fornecidas algumas observações sobre o domínio do problema escolhido e sobre o processo de desenvolvimento do sistema. Em seguida, as Seções 4.3, 4.4 e 4.5 descrevem o *design* resultante da primeira, segunda e terceira iterações de desenvolvimento, respectivamente, destacando também alguns compromissos envolvidos e algumas observações peculiares ao desenvolvimento em DSOT.

4.2 Observações Gerais Sobre o Processo de Desenvolvimento e o Domínio do Problema

Para o estudo de caso, foi adotado um processo de desenvolvimento iterativo para criar um sistema de software do domínio de sistemas de edição de texto. Nesta seção é justificada a escolha de um processo iterativo (Subseção 4.2.1) e comenta-se sobre os critérios de priorização de requisitos possíveis para cada iteração (Subseção 4.2.2). Também são mencionados alguns materiais que contribuiram para a análise do problema (Subseção 4.2.3) e compara-se brevemente o domínio do problema escolhido com outros domínios em termos de complexidade do desenvolvimento de software (Subseção 4.2.4).

4.2.1 Motivação para dividir o projeto em iterações

O projeto de desenvolvimento do sistema de software descrito neste capítulo foi dividido em iterações. A motivação para tal divisão é combater o risco de exceder a capacidade cognitiva disponível para se lidar com complexidade arbitrária. Um sistema de software é um produto intelectualmente complexo, portanto convém aplicar as estratégias de *dividir para conquistar* e de *separação de interesses* (Dijkstra, 1976) para dividir a complexidade total em partes menos complexas, e depois integrar a solução das partes para se formar o sistema total. A diretriz utilizada para reduzir o risco de fracasso em lidar com a complexidade é que o objetivo de uma iteração deve ser o desenvolvimento de um conjunto relativamente pequeno de características, em um intervalo de tempo relativamente pequeno. Para se reduzirem os riscos, um incremento deve ser estabelecido ao início de cada iteração como um conjunto de requisitos cujos detalhes de *design* sejam intelectualmente gerenciáveis de forma confortável ao longo daquela iteração. As outras características exigidas para formar o sistema completo são desenvolvidas em iterações subsequentes.

Efetuam-se tantas iterações quanto forem necessárias para se concluir o sistema, ou até que os recursos do projeto acabem. O principal resultado de uma iteração é um *protótipo executável*, que consiste em um programa (ou um conjunto de programas) que incorpora parte das características que o sistema de software completo deve possuir ao final do projeto. O protótipo executável obtido em uma iteração contém a implementação de um superconjunto das características contidas na *versão* anterior. A versão do protótipo executável de uma iteração distingue-se das demais por um *identificador de versão*, normalmente um número ou uma letra. A diferença global entre uma versão do protótipo e a versão obtida na iteração imediatamente anterior é denominada *incremento*. O incremento contém, dentre outras coisas, os módulos adicionados, as alterações feitas nos módulos anteriores, e as características de alto nível que diferenciam uma determinada versão da versão imediatamente anterior.

A cada iteração, o projeto progride um pouco. Os sinais deste progresso são a evolução estabelecida pelo surgimento de uma nova versão do protótipo, a criação de documentação final, a criação de protótipos descartáveis para esclarecer algumas questões de *design*, entre outras.

Efetuando-se uma análise com base no protótipo executável obtido em cada iteração, o processo dentro de uma iteração parece ter sido linear: ocorre o desenvolvimento de todos os testes de uma classe A, depois a classe A com todas as rotinas, depois todos os testes de outra classe B, em seguida todas as rotinas de B e assim por diante, até que o protótipo da iteração esteja criado. Na prática, contudo, o processo dentro de uma iteração não é sequencial; ele na verdade é mais caótico, não no sentido de indisciplinado, mas no sentido de que é oportunístico e orientado pela criatividade. Durante a iteração, criam-se alguns testes para algumas rotinas de uma classe A, então se codificam as rotinas de A, depois se criam testes para algumas rotinas de uma classe B, descobre-se a necessidade de novas rotinas para A, que então são criadas, eventualmente se descobre que é possível melhorar a solução obtida até agora e então uma refatoração é efetuada, cria-se um protótipo descartável para experimentar uma idéia e assim por diante, até que a iteração esteja completa. Neste capítulo é enfatizada a descrição das características de *design* dos protótipos executáveis obtidos em cada iteração, comentando em alguns pontos no meio da descrição algumas questões notáveis enfrentadas durante o processo de desenhar o software.

4.2.2 Critério de priorização de requisitos

Quando os requisitos de um sistema são definidos, mesmo que imprecisamente, é provável que a quantidade de requisitos definidos será superior ao número de requisitos que podem ser implementados em uma iteração de duração relativamente pequena. Assim, no início de uma iteração o desenvolvedor é levado a adotar um critério para selecionar os requisitos que serão satisfeitos naquela iteração. Vários critérios de priorização de requisitos podem ser adotados, como: requisitos de maior valor para o usuário primeiro; requisitos que causam maior impacto na arquitetura do sistema primeiro; requisitos que acarretam maior risco e maior complexidade primeiro; e requisitos mais convenientes para o desenvolvedor primeiro. À exceção do primeiro critério, os demais resultam em diferenças muito pequenas uns dos outros quando aplicados.

Para este projeto, adotou-se o critério de conveniência para o desenvolvedor: são satisfeitos primeiro os requisitos que o desenvolvedor acredita que irão facilitar o desenvolvimento subsequente. Por um lado, isso é vantajoso porque permite reduzir a quantidade e a severidade de eventuais mudanças invasivas sobre o protótipo executável produzido na iteração anterior. Por outro lado, este critério não é muito realista para um projeto de desenvolvimento de sistemas de software comerciais: em geral, o desenvolvimento de um sistema comercial tende a ser mais centrado no usuário. Será mais conveniente ao usuário utilizar como critério de prioridade o valor de cada característica requerida: as características de maior valor agregado têm prioridade na implementação, e são escolhidas tantas características quanto necessárias para o esforço de uma iteração. A estratégia mais adequada no desenvolvimento de sistemas de software comerciais típicos é deixar a

escolha ser feita pelo usuário, com a assistência do desenvolvedor, em um processo de negociação cujo objetivo é que as duas partes sejam beneficiadas no final. Levando esse cenário em consideração, é importante haver pesquisas futuras que analisem o que ocorre com o *design* do sistema ao longo das iterações de um projeto de desenvolvimento de software comercial: um programa orientado a temas seria realmente menos sensível às mudanças do que, por exemplo, um programa orientado a objetos, caso os dois projetos utilizem o critério de priorizar os requisitos que possuem maior valor ao usuário? Em quais condições?

4.2.3 Materiais que contribuiram para a análise do problema

Uma atividade importante para o início do projeto é consultar a literatura do domínio do problema para: obter idéias para a análise do problema, o *design* da solução e avaliação de qualidade do produto; encontrar módulos reutilizáveis; e outros possíveis benefícios. Para o sistema *Simple Notepad*, foram consultados os materiais a seguir.

O domínio de edição de textos está bem fundamentado em Meyrowitz e van Dam (1982). Há também um livro que complementa aquele material, discutindo alguns pontos importantes no desenvolvimento de editores de texto (Finseth, 1999).

Para a linguagem de programação Java, há uma Interface de Programação de Aplicativos que forneceu alguma inspiração para o desenvolvimento do sistema aqui descrito (Sun Microsystems, 2005). Várias classes foram sujeitas a uma observação mais cautelosa, em particular as classes e interfaces *JTextComponent*, *Caret*, e *Document* do pacote *javax.swing.text*.

O software de fonte aberta (*open source*) *JEdit* é um editor de textos bastante conhecido por programadores, especialmente desenvolvedores *Java* (Pestov, 2005). O estudo de seu *design* também foi considerado para se desenvolver o sistema de edição de textos. Em particular, foi mais cautelosamente estudada a classe *org.gjs.sp.jedit.textarea.JEditTextArea*, que implementa o núcleo do editor.

Há também um *Wiki* em constante evolução que fornece informações gerais sobre sistemas de edição de texto, incluindo desenvolvimento, uso e comparação entre sistemas diferentes (Perrella, 2005). Várias outras páginas *Web* forneceram definições e comentários que contribuíram de alguma forma para definir o processo de desenvolvimento do sistema de edição de textos (Kelly, 2005; UNIX Help, 2005).

4.2.4 Características do domínio escolhido em comparação com outros domínios

Um sistema de informação é um tipo de software que possui em geral poucas demandas de processamento e muitas demandas de armazenamento e recuperação de dados. Assim, um modelo de estrutura da informação para um sistema de informação é geralmente muito rico em conceitos, o que torna a análise do problema mais difícil. Por outro lado, sistemas de informação são exemplos

de software nos quais o domínio de problemas corresponde grosseiramente a uma perspectiva do "mundo real", o que tornaria a análise do problema mais fácil.

Para alguns tipos de software, tais como compiladores, sistemas operacionais, e sistemas de edição de textos, o domínio de problemas não é povoado por conceitos do "mundo real", mas por conceitos de software. Isso torna a análise do problema mais difícil, já que deve-se ter cuidado extra para não confundir *conceitos* de software, do problema, e *objetos* de software, da solução orientada a objetos ou orientada a temas.

Um sistema de edição de textos é um tipo de software que possui demandas descomplicadas de armazenamento e recuperação de dados, e relativamente poucas demandas de processamento. Portanto, é razoável que um modelo de estrutura da informação para um sistema de edição de textos não represente muitos conceitos relevantes, embora deva se tomar cuidado com a questão de que os conceitos envolvidos são de software.

O modelo de estrutura da informação de sistemas de edição de textos ajuda a explicar porque os usuários consideram tais sistemas de software simples. Há poucos conceitos relevantes envolvidos, e esses conceitos não possuem muitas propriedades. Mas, se for observado o diagrama de atividades de alguma característica maior, perceber-se-á que há vários detalhes envolvidos, que tornam tais sistemas razoavelmente complexos. Usuários os acham simples, mas desenvolvedores não, ao menos em situações nas quais não é possível reutilizar módulos que contêm a implementação de grande parte das funcionalidades necessárias.

4.3 Primeira Iteração

Os requisitos escolhidos para a primeira iteração de desenvolvimento do sistema de edição de textos *Simple Notepad* tratam de permitir a edição básica de um texto descartável, sem a possibilidade de salvar, imprimir, fazer seleção de fragmentos do documento, e outras funcionalidades interessantes para um editor de texto. Eles são os seguintes:

1. A estrutura da informação tratada pelo programa é percebida pelo usuário conforme o seguinte modelo:

   1 . 1 O documento é composto por linhas de caracteres.

   1 . 2 Todas as operações de edição sobre o documento são efetuadas de acordo com um cursor que é composto por duas posições, uma linha e uma coluna.

2. Uma área de texto controla as operações de edição que alteram o estado do documento. A entrada é efetuada por teclado sobre a área de texto como se segue:

   2 . 1 qualquer caractere imprimível é digitado: o caractere é inserido no documento conforme a posição atual do cursor e o cursor é movido para a direita;

2 . 2 a tecla $ENTER$ é pressionada: uma quebra de linha é criada no documento de acordo com a posição atual do cursor;

2 . 3 a tecla $HOME$ é pressionada: o cursor é movido para o início da linha;

2 . 4 a tecla END é pressionada: o cursor é movido para o fim da linha;

2 . 5 a tecla $\uparrow$ é pressionada: o cursor é movido para cima;

2 . 6 a tecla $\downarrow$ é pressionada: o cursor é movido para baixo;

2 . 7 a tecla $\leftarrow$ é pressionada: o cursor é movido para a esquerda;

2 . 8 a tecla $\rightarrow$ é pressionada: o cursor é movido para a direita;

2 . 9 a tecla $DELETE$ é pressionada: a próxima linha é concatenada com a linha atual se o cursor estiver no fim da linha, caso contrário o próximo caractere é removido;

2 . 10 a tecla $BACK\_SPACE$ é pressionada: a linha atual é concatenada com a linha anterior se o cursor estiver no começo da linha, caso contrário o caractere anterior é removido e o cursor é movido para a esquerda.

3. Uma área de texto fornece uma visão para o estado atual do documento. Qualquer atualização do estado do documento é percebida como saída na área de texto. Além do mais, o retorno na área de texto é "imediato" quando o documento muda.

4. As áreas de texto de entrada (req. 2) e de saída (req. 3) são uma só.

Na primeira iteração deste projeto, coincidentemente, os requisitos selecionados são os mesmos requisitos que seriam priorizados caso fosse adotado o critério de maior valor ao usuário primeiro. Talvez isso ocorra porque, ao início de um projeto, há uma ordem "natural" para se implementar os requisitos: deve ser desenvolvida primeiro a "base fundamental" do programa, que corresponde às suas características mais essenciais, para que se possa então satisfazer outros requisitos menos "fundamentais" (mais nem por isso pouco importantes) nas evoluções subsequentes do sistema. Seria interessante investigar futuramente se este fenômeno ocorre na maioria dos domínios, pois uma eventual recorrência deste fenômeno poderia sugerir aos pesquisadores de engenharia de requisitos de software diretrizes para criar métodos de engenharia de requisitos que se beneficiem dessa recorrência.

Para a primeira iteração de desenvolvimento do sistema *Simple Notepad*, foi desenvolvido inicialmente o modelo de estrutura da informação mostrado na Figura 4.1, conforme o requisito 1. Esse modelo indica basicamente que um documento é formado por linhas de caracteres e agrega um cursor. Essa não seria a única maneira de fazê-lo; o modelo poderia ter sido diferente, como por exemplo aquele exposto na Figura 4.2, que indica que um documento é formado por vários elementos, que podem ser *strings* (fragmentos de texto contíguo) ou quebras de linha (elementos que indicam que uma nova linha está prestes a iniciar). A escolha do modelo causa um impacto profundo no *design* final do sistema.

No primeiro modelo é explicitamente reconhecida a alta frequência da execução de operações que envolvem mais de uma linha. No segundo modelo, por outro lado, são privilegiadas somente as operações que envolvem caracteres vizinhos. Por este motivo o segundo modelo foi preterido em favor do primeiro.

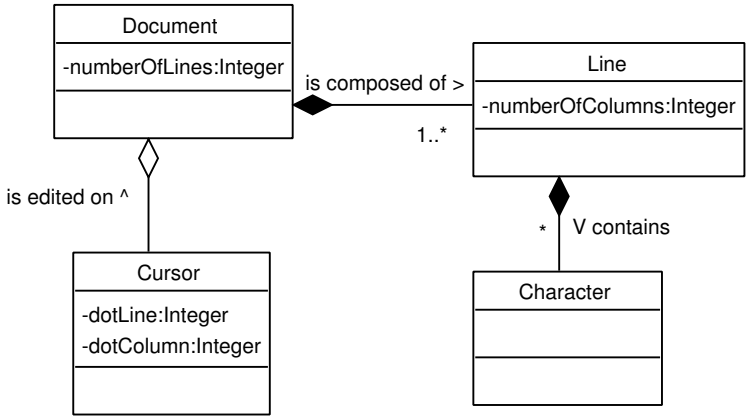


Figura 4.1: Modelo de estrutura da informação do *Simple Notepad* para a primeira iteração.

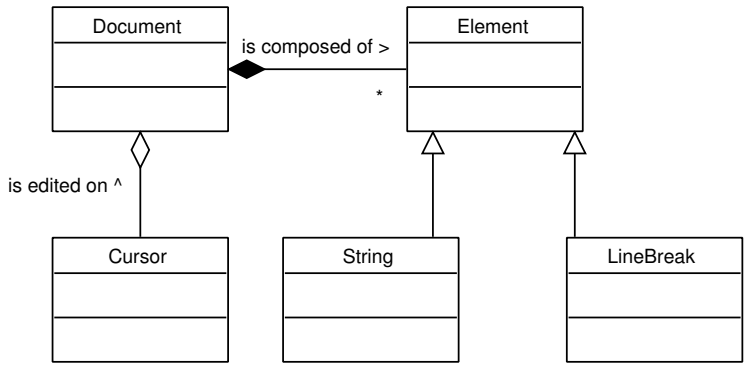


Figura 4.2: Como poderia ter sido o modelo de estrutura da informação escolhido para o *Simple Notepad*.

Na primeira iteração, definiu-se que o protótipo executável resultante seria composto por uma janela que contém uma área de texto. As edições do documento seriam feitas sobre a área de texto, enquanto a janela teria as capacidades de redimensionamento e fechamento do programa. Considerou-se, então, que o programa seria composto por dois temas: **CompleteWindowSetup**, responsável pelas características da janela e da inserção da área de texto na janela, e **TextArea-MVC**, responsável pela implementação das funcionalidades referentes à área de texto. A fórmula de composição que indica como o programa é formado é mostrada a seguir, e o diagrama correspondente àquela fórmula de composição é exibido na Figura 4.3.

```
CompleteWindowSetup := ...

TextAreaMVC := ...

SimpleNotepadApp := CompleteWindowSetup + TextAreaMVC;
```

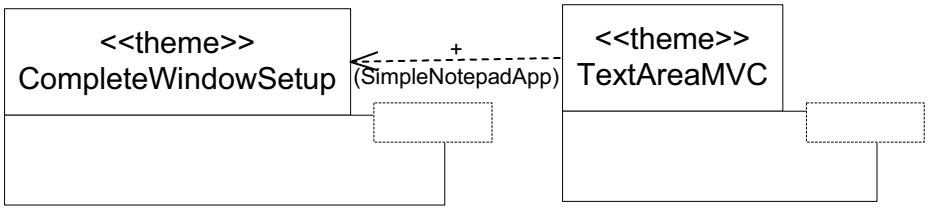


Figura 4.3: Diagrama de temas indicando como o programa é gerado.

A estruturação dos subsistemas do programa em torno de padrões arquiteturais relativamente estáveis facilita a evolução de software. A primeira grande vantagem de se separar o programa nos dois temas mencionados é a relativa estabilidade do *design*: novas características para as próximas iterações poderiam ser inseridas sem ter de afetar a estrutura maior do programa. Outras vantagens importantes são a expressividade, a compreensibilidade e a explicitação da estrutura do programa. A regra de composição mencionada indica que o programa é formado por uma janela e uma área de texto. Dificilmente seria possível tanta concisão em uma solução orientada a objetos.

Mas como é a semântica da composição entre os temas **CompleteWindowSetup** e **TextAreaMVC**? Para responder a essa pergunta, é necessário observar como cada tema está estruturado internamente.

4.3.1 Tema CompleteWindowSetup

O tema **CompleteWindowSetup** é formado pelas classes *SimpleNotepad*, *SimpleTextArea* e *ApplicationExitWindowListener*. Os principais trechos do código-fonte das três classes são mostrados a seguir. O diagrama que mostra as três classes é apresentado na Figura 4.4.

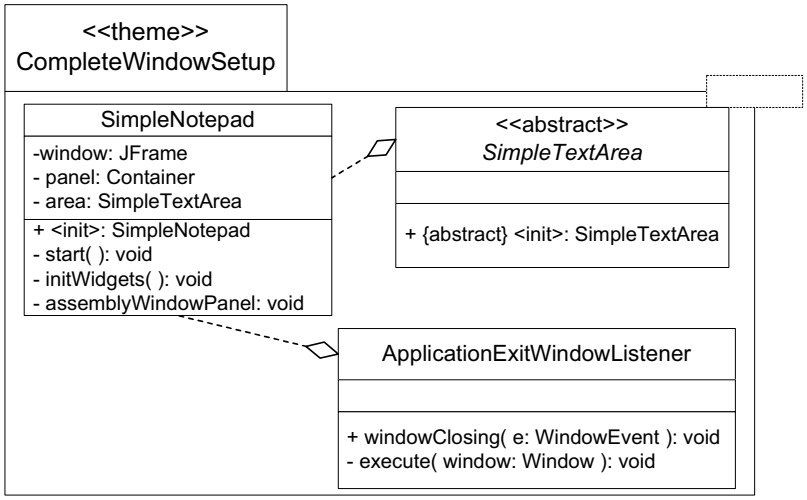


Figura 4.4: Diagrama de classes do tema **CompleteWindowSetup**.

```
public class SimpleNotepad{
    static public void main( String[] args ){
        new SimpleNotepad( ).start( );
    }

    public SimpleNotepad( ){
        this.initWidgets( );
        this.assemblyWindowPanel( );
    }

    public void start( ){
        this.window.pack( );
        this.area.requestFocusInWindow( );
        this.window.setSize( 550, 500 );
        this.window.setVisible( true );
    }

    private void initWidgets( ){
        this.window = new JFrame( "Simple Notepad" );
        this.window.setDefaultCloseOperation( JFrame.DO_NOTHING_ON_CLOSE );
        this.window.addWindowListener( new ApplicationExitWindowListener( ) );
        this.panel = this.window.getContentPane( );
        this.area = new SimpleTextArea( ); /* instantiation of an abstract class */
    }

    private void assemblyWindowPanel( ){
        this.panel.setLayout( new BorderLayout( ) );
        this.panel.add( this.area, BorderLayout.CENTER );
    }

    private Container panel;
    private JFrame window;
    private SimpleTextArea area;
}
```

```
abstract public class SimpleTextArea extends JPanel {
    abstract public SimpleTextArea( );
}
```

```
public class ApplicationExitWindowListener extends WindowAdapter{
    public void windowClosing( WindowEvent e ){
        this.execute( e.getWindow( ) );
    }

    private void execute( Window window ){
            window.setVisible( false );
            window.dispose( );
            System.exit( 0 );
    }
}
```

Basicamente, a classe *SimpleNotepad* instancia uma janela e uma área de texto (*SimpleTextArea*), e insere a área de texto na janela. A classe *ApplicationExitWindowListener* apenas implementa a política de fechamento da aplicação iniciada por *SimpleNotepad*.

Observe que a classe *SimpleTextArea* é abstrata, e ao mesmo tempo está sendo instanciada na implementação de *SimpleNotepad*! Isso é válido no DSOT, apesar de não sê-lo no POO. A classe *SimpleTextArea* é abstrata no tema **CompleteWindowSetup** porque a implementação dela não é essencial àquele tema. O propósito de **CompleteWindowSetup** é apenas estabelecer a configuração dos elementos gráficos do programa. A implementação de *SimpleTextArea* está contida em outro tema, no caso o **TextAreaMVC**. De qualquer forma, uma implementação abstrata de *SimpleTextArea* está presente no tema **CompleteWindowSetup** para satisfazer a restrição de completeza declarativa.

Quando **CompleteWindowSetup** é composto com **TextAreaMVC** para formar o tema final **SimpleNotepadApp**, a classe *SimpleTextArea* de **CompleteWindowSetup** é complementada com a implementação da classe *SimpleTextArea* de **TextAreaMVC**. Em particular, espera-se que o construtor de *SimpleTextArea*, abstrato em **CompleteWindowSetup**, esteja implementado em **TextAreaMVC**.

4.3.2 Tema TextAreaMVC

Seguindo a diretriz de se estruturar software por meio de padrões arquiteturais relativamente estáveis, convém estruturar a implementação da área de texto conforme o padrão *Model-View-Controller* (MVC). A adoção desse padrão arquitetural torna a solução bastante compreensível porque há uma divisão clara de responsabilidades entre o modelo, a visão e o controlador. As vantagens do MVC ficarão ainda mais claras na discussão da segunda iteração, na qual se optou por adicionar à área de texto características e funcionalidades que envolvem intervalo de seleção.

O modelo da área de texto contempla a parte da implementação independente da interface gráfica com o usuário, correspondendo ao núcleo de computação da área de texto. A visão constitui

a parte da interface gráfica da área de texto que controla o processamento de saída ao usuário, correspondendo à implementação dos algoritmos que fornecem retorno ao usuário com relação ao estado do documento. Por sua vez, o controlador constitui a parte da interface gráfica da área de texto que controla o processamento da entrada do usuário, correspondendo à implementação dos algoritmos que processam comandos do usuário e os convertem em chamadas de rotinas que agem sobre o documento.

A interação entre os três elementos da arquitetura ocorre de uma maneira peculiar. A visão é registrada como observadora das mudanças de estado no modelo, fazendo com que em efeito o modelo avise a visão tão logo o estado é alterado. Quando o usuário envia um comando para o *widget*, o controlador processa o comando e normalmente solicita uma mudança de estado no modelo. O modelo, em seguida, efetua as computações necessárias para a mudança de estado. Assim que isto é concluído, o modelo avisa a visão, que efetua as computações necessárias para mostrar ao usuário a alteração de estado no modelo.

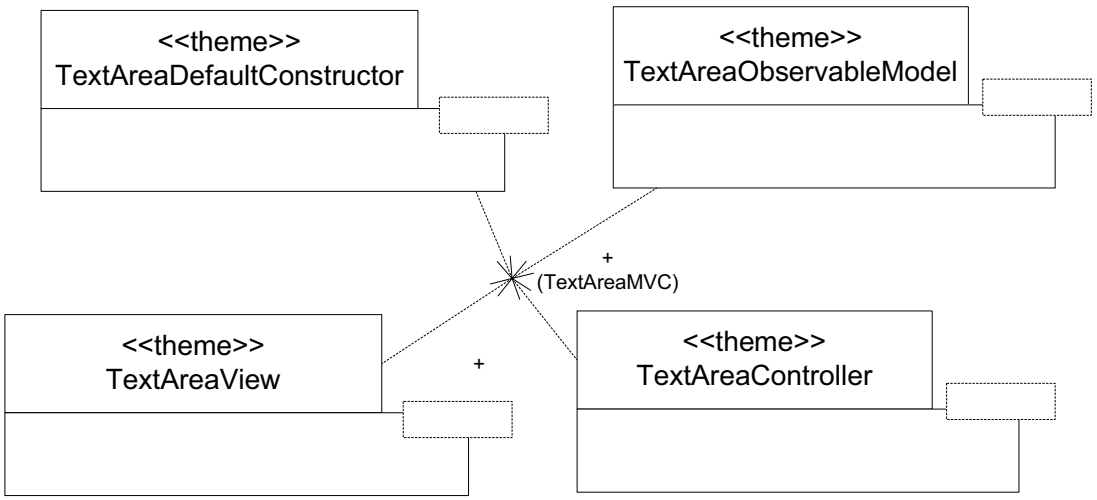


Figura 4.5: Composição que gera o tema **TextAreaMVC**

Para gerar o tema **TextAreaMVC** foram desenhados quatro temas: **TextAreaObservable-Model**, **TextAreaView**, **TextAreaController** e **TextAreaDefaultConstructor**. A composição desses temas é mostrada na Figura 4.5. Cada tema é comentado separadamente nas subseções a seguir.

4.3.3 Tema TextAreaView

A visão da área de texto é implementada no tema **TextAreaView**, formado basicamente pelas classes *SimpleTextArea* e *Document*. O objetivo desse tema é satisfazer o requisito 3. O diagrama de classes correspondente ao tema **TextAreaView** é mostrado na Figura 4.6.

A classe *SimpleTextArea* é o núcleo do tema **TextAreaView**. A rotina paintComponent (Graphics) contém a implementação do algoritmo que desenha as linhas de caracteres e o cursor no painel

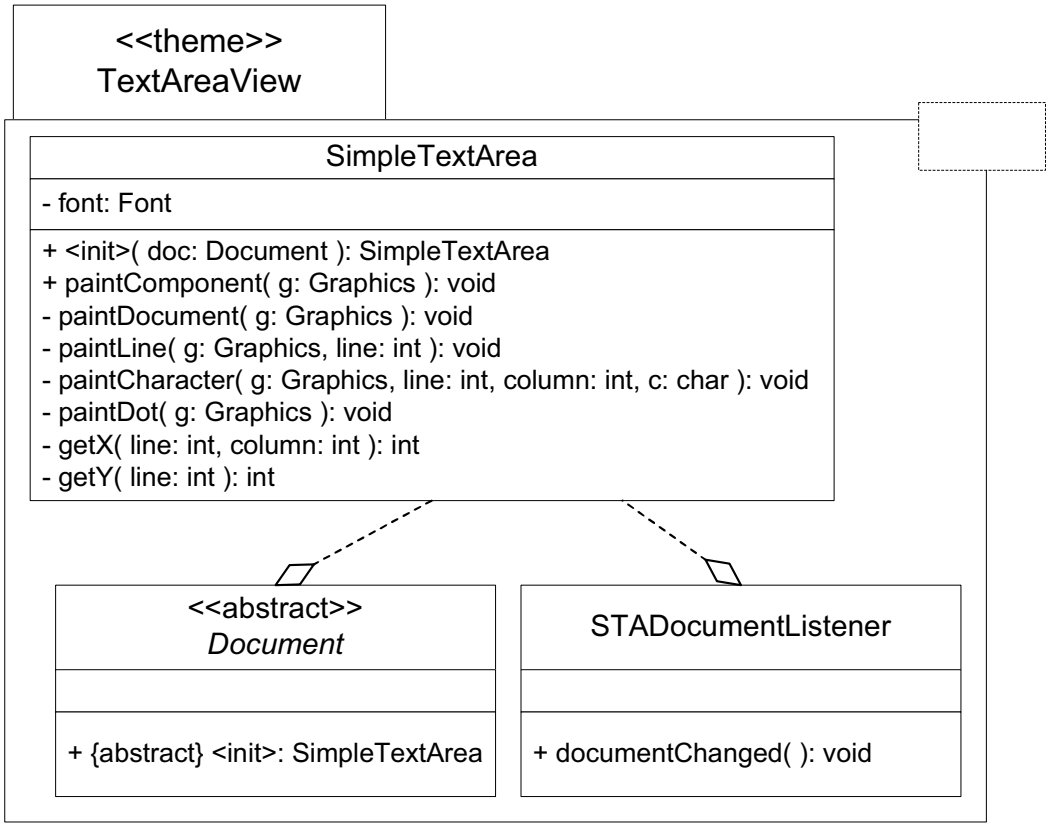


Figura 4.6: Diagrama de classes do tema **TextAreaView**.

da área de texto. Observe que a classe *Document* está presente no tema **TextAreaView** apenas para satisfazer a restrição de completeza declarativa.

Eventualmente um protótipo descartável é criado sob demanda, conforme a necessidade do desenvolvedor. Um protótipo descartável é um modelo executável (isto é, implementado) que é útil para experimentar uma idéia ou esclarecer uma questão de *design*, e que pode ser descartado após ter sido atingido o propósito que motivou a sua criação. A implementação do tema *View* ilustra a idéia de como utilizar protótipos descartáveis para auxiliar o processo de *design* de software. Foi criado um protótipo descartável para se descobrir a fórmula para desenhar corretamente as linhas de caracteres nos eixos x e y do painel da área de texto. Segue o código do protótipo mencionado.

```
public class LineAndCharacterDrawingAlgorithm extends JPanel{ //JComponent
    static public void main( String[] args ){
        JMenuBar bar = new JMenuBar( );
        bar.add( new JMenu( "Edit" ) );
        JFrame f = new JFrame( "Simple Notepad" );
        f.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        f.setJMenuBar( bar );
        f.setLayout( new BorderLayout( ) );
        f.add( new JScrollPane( new LineAndCharacterDrawingAlgorithm( ),
         VERTICAL_SCROLLBAR_ALWAYS, HORIZONTAL_SCROLLBAR_ALWAYS ),
         BorderLayout.CENTER );
        f.setSize( 550, 500 );
        f.setVisible( true );
    }

    public LineAndCharacterDrawingAlgorithm( ){
        super.setBackground( Color.WHITE );
        Font f = new Font( "Monospaced", Font.PLAIN, 12 );
        super.setFont( f );
        this.metrics = super.getFontMetrics( f );
    }

    public void paint( Graphics g ){
        super.paint( g );
        int x = 0;
        int y = this.metrics.getHeight( );
        g.drawString( "a", x, y );
        x += this.metrics.charWidth( 'a' );
        g.drawString( "b", x, y );
        x += this.metrics.charWidth( 'b' );
        g.drawString( "c", x, y );
        g.drawLine( x, y - this.metrics.getAscent( ), x, y +
         this.metrics.getDescent( ) );

        x = 0;
        y += this.metrics.getHeight( );
        g.drawString( "d", x, y );
        x += this.metrics.charWidth( 'd' );
        g.drawString( "e", x, y );
        x += this.metrics.charWidth( 'e' );
        g.drawString( "f", x, y );
        x += this.metrics.charWidth( 'f' );
        g.drawString( "\t", x, y );
        x += this.metrics.charWidth( '\t' );
        g.drawString( "g", x, y );
        x += this.metrics.charWidth( 'g' );
        g.drawRect( x, y - this.metrics.charWidth( '#' ),
         this.metrics.charWidth( '#' ), this.metrics.charWidth( '#' ) );
        x += this.metrics.charWidth( '#' );
    }

    private FontMetrics metrics;
}
```

4.3.4 Tema TextAreaController

O controlador da área de texto é implementado no tema **TextAreaController**, responsável por implementar o requisito 2 completo. A classe *SimpleTextArea*, as várias classes *Action*s e a classe *Document* formam esse tema. Como o requisito 2 é dividido em vários subrequisitos, considerou-se definir o tema **TextAreaController** como o resultado da composição de vários temas menores, sendo que cada tema contém uma implementação que satisfaz um dos subrequisitos. Segue-se a composição que gera o tema **TextAreaController**.

```
TextAreaController := InsertCharacter + CreateLine + MoveCursorDown +
 MoveCursorLeft + MoveCursorRight + MoveCursorToEnd + MoveCursorToHome +
 MoveCursorUp + RemovePreviousCharacterOrAppendCurrentToPreviousLine +
 RemoveNextCharacterOrAppendNextToCurrentLine + KeyboardFocusability{
    SimpleTextArea.<init>( ) => <init>( Document doc );
};
```

Os temas que compõem **TextAreaController** são estruturalmente muito parecidos. Observe a seguir, por exemplo, o código do tema **CreateLine**, cuja implementação satisfaz o subrequisito 2.2.

```
abstract public class SimpleTextArea{
    public SimpleTextArea( Document doc ){
        Action createLineAction = new CreateLineAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "pressed ENTER" ),
         createLineAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( createLineAction.getValue( Action.NAME ),
         createLineAction );
    }

    abstract private InputMap getInputMap( );

    abstract private ActionMap getActionMap( );
}
```

```
abstract public class Document{
    abstract public void createLineBreak( );

    abstract public void moveCursorDown( )
     throws IllegalStateException;

    abstract public void moveCursorToHome( );
}
```

```
public class CreateLineAction extends AbstractAction{
    public CreateLineAction( Document doc )
     throws NullPointerException{
        super( "Create line" );

        if( doc == null )
            throw new NullPointerException( );

        this.doc = doc;
        this.updateEnabling( );
    }

    public void actionPerformed( ActionEvent e )
     throws IllegalStateException{
        if( !this.isEnabled( ) )
            throw new IllegalStateException( );

        this.doc.createLineBreak( );
        this.doc.moveCursorToHome( );
        this.doc.moveCursorDown( );
    }

    public void setEnabled( boolean enabled )
     throws UnsupportedOperationException{
        throw new UnsupportedOperationException( );
    }

    private void updateEnabling( ){
        super.setEnabled( true );
    }

    private Document doc;
}
```

Algumas rotinas ao mesmo tempo abstratas e privadas aparecem na classe *SimpleTextArea* do tema **CreateLine**. No POO, a criação de uma rotina ao mesmo tempo abstrata e privada em uma classe não tem sentido. Mas no DSOT isso é válido, e indica que a rotina deve ser composta com outra rotina, que pode ser privada ou até mesmo pública, para se tornar completa.

A implementação do construtor de *SimpleTextArea* indica que uma entrada do usuário (no caso do tema **CreateLine**, o ato de pressionar a tecla $ENTER$) é convertida no disparo de um evento para uma *Action* (no caso do tema **CreateLine**, uma *CreateLineAction*), que então irá atender a solicitação do usuário e alterar o estado do modelo (representado pela classe *Document*).

Com a exceção dos temas **InsertCharacter** (que satisfaz o subrequisito 2.1) e **KeyboardFocusability**, os temas que compõem **TextAreaController** são muito parecidos com o **CreateLine** em termos de estrutura. A implementação da classe *SimpleTextArea* do tema **MoveCursorDown**, cuja implementação satisfaz o subrequisito 2.6, ilustra esta semelhança.

```
abstract public class SimpleTextArea{
    public SimpleTextArea( Document doc ){
        Action moveCursorDownAction = new MoveCursorDownAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "pressed DOWN" ),
         moveCursorDownAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( moveCursorDownAction.getValue( Action.NAME ),
         moveCursorDownAction );
    }

    abstract private InputMap getInputMap( );

    abstract private ActionMap getActionMap( );
}
```

A classes do tema **InsertCharacter** possui uma implementação estruturalmente diferente das respectivas classes dos demais temas que compõem **TextAreaController**, como apresentado na Figura 4.7.

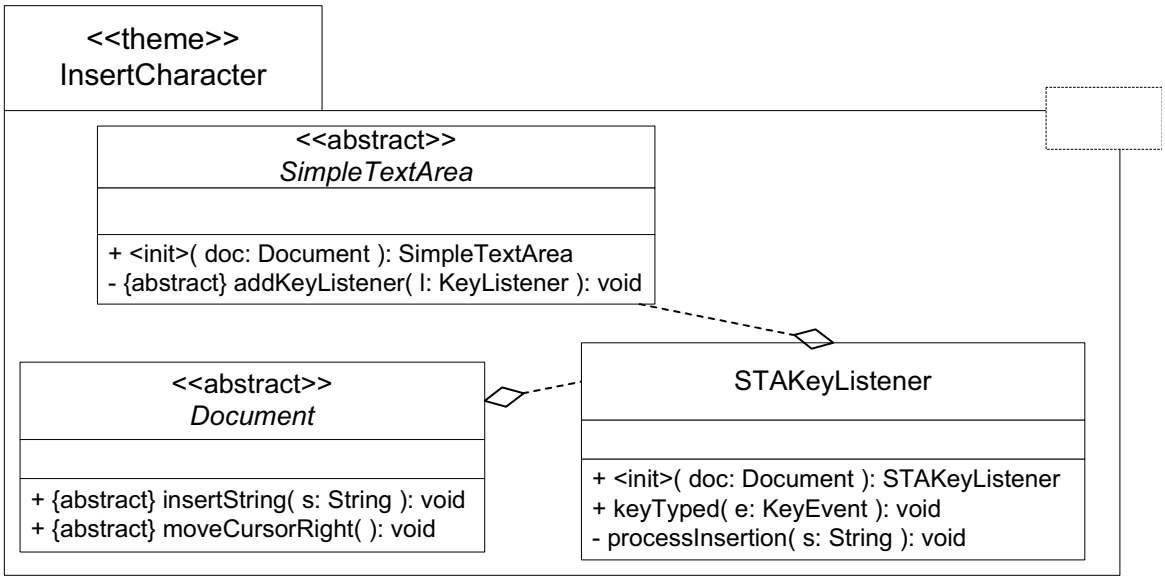


Figura 4.7: Diagrama de classes do tema **InsertCharacter**.

O tema **KeyboardFocusability** é responsável por fazer com que a área de texto aceite comandos do usuário por meio do teclado, isto é, torna a área de texto focável por teclado. Sua implementação aparece a seguir.

```
abstract public class SimpleTextArea{
    public SimpleTextArea( ){
        this.setFocusable( true );
        this.addFocusListener( new FocusAdapter( ){} );
    }

    abstract private void setFocusable( boolean focusable );

    abstract private void addFocusListener( FocusListener l );
}
```

Na fórmula de composição que gera o tema **TextAreaController**, mostrada a seguir novamente, aparece a primeira redefinição em uma fórmula de composição. A classe *SimpleTextArea* do tema **KeyboardFocusability** possui um construtor que não recebe parâmetros. Deseja-se convertê-lo em um construtor que recebe um parâmetro do tipo *Document*, para que este construtor possa ser composto corretamente no resultado final. Isto é feito por meio de uma redefinição do construtor.

```
TextAreaController := InsertCharacter + CreateLine + MoveCursorDown +
 MoveCursorLeft + MoveCursorRight + MoveCursorToEnd + MoveCursorToHome +
 MoveCursorUp + RemovePreviousCharacterOrAppendCurrentToPreviousLine +
 RemoveNextCharacterOrAppendNextToCurrentLine + KeyboardFocusability{
    SimpleTextArea.<init>( ) => <init>( Document doc );
};
```

Já que foram mostrados os temas primitivos que compõem o tema **TextAreaController**, torna-se possível discutir concretamente a semântica da composição. Os temas são formados internamente por classes. Quando dois temas são compostos, o resultado da composição depende das classes que formam cada tema. As classes que são consideradas diferentes devem logicamente aparecer intactas no tema resultante. As classes que são consideradas as mesmas (apesar de estarem descritas sob perspectivas diferentes em temas diferentes) devem se fundir em uma só. Para cada uma dessas classes, os atributos e as rotinas que existem somente em um dos dois temas primitivos devem logicamente aparecer intactas no tema resultante. Os atributos que se repetem nos duas classes correspondentes dos dois temas devem aparecer somente uma vez no tema resultante. Para rotinas que existem em mais de um tema primitivo, o resultado da composição aparece conforme a semântica da composição exposta no Capítulo 3.

A classe *SimpleTextArea* do tema **TextAreaController**, que resulta da combinação das classes *SimpleTextArea* dos temas primitivos, é listada a seguir. Perceba que a implementação do construtor é de fato a implementação combinada dos construtores nos temas primitivos.

```
abstract public class SimpleTextArea{
    public SimpleTextArea( Document doc ){
        this.addKeyListener( new STAKeyListener( doc ) );
        Action createLineAction = new CreateLineAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "pressed ENTER" ),
         createLineAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( createLineAction.getValue( Action.NAME ),
         createLineAction );
        Action moveCursorDownAction = new MoveCursorDownAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "pressed DOWN" ),
         moveCursorDownAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( moveCursorDownAction.getValue( Action.NAME ),
         moveCursorDownAction );
        Action moveCursorLeftAction = new MoveCursorLeftAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "pressed LEFT" ),
         moveCursorLeftAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( moveCursorLeftAction.getValue( Action.NAME ),
         moveCursorLeftAction );
        Action moveCursorRightAction = new MoveCursorRightAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "pressed RIGHT" ),
         moveCursorRightAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( moveCursorRightAction.getValue( Action.NAME ),
         moveCursorRightAction );
        Action moveCursorToEndAction = new MoveCursorToEndAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "pressed END" ),
         moveCursorToEndAction.getValue( Action.NAME ) );
        ... //other similar code
        this.setFocusable( true );
        this.addFocusListener( new FocusAdapter( ){} );
    }

    abstract private void addKeyListener( KeyListener l );

    abstract private void setFocusable( boolean focusable );

    abstract private void addFocusListener( FocusListener l );

    abstract private InputMap getInputMap( );

    abstract private ActionMap getActionMap( );
}
```

A classe *Document* do tema **TextAreaController** também é resultado da combinação das demais classes *Document*, conforme aparece na Figura 4.8.

4.3.5 Tema TextAreaModel

O tema **TextAreaModel** é responsável pela implementação do modelo da área de texto. Para o programa *Simple Notepad*, o principal objetivo do tema **TextAreaModel** é conter a implementação das rotinas da classe *Document* que não estão implementadas nos temas **TextAreaView** e **TextAreaController**. Nessa implementação de **TextAreaModel**, optou-se por criar outras duas classes para dividir as responsabilidades e facilitar o *design* da classe *Document*: *Line* e *Cursor*. Na Figura 4.9 aparece a organização geral das três classes.

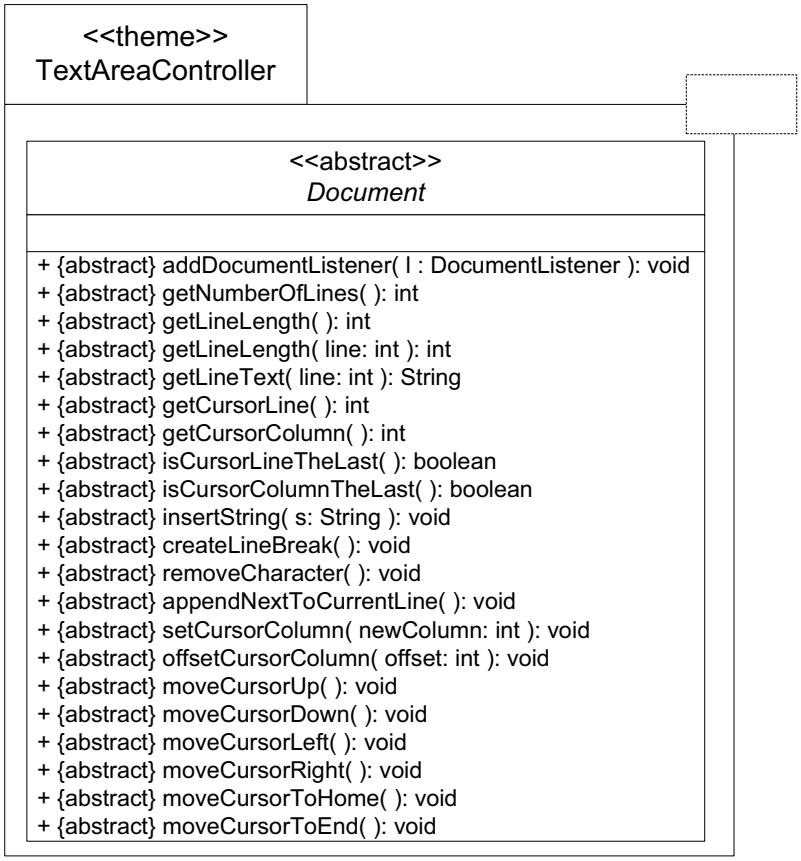


Figura 4.8: Classe *Document* do tema **TextAreaController**.

A funcionalidade básica do tema **TextAreaModel** tem de ser complementada para permitir que a visão monitore as alterações de estado no modelo para fornecer um retorno acurado ao usuário. Para isso, foram criados os temas **Monitoring** e **AbstractDocumentMonitoring**.

O tema **Monitoring** contém uma implementação reutilizável do padrão de *design Observation* ou *Observer* (Gamma et al., 1995). Sua implementação, bastante simples, aparece a seguir.

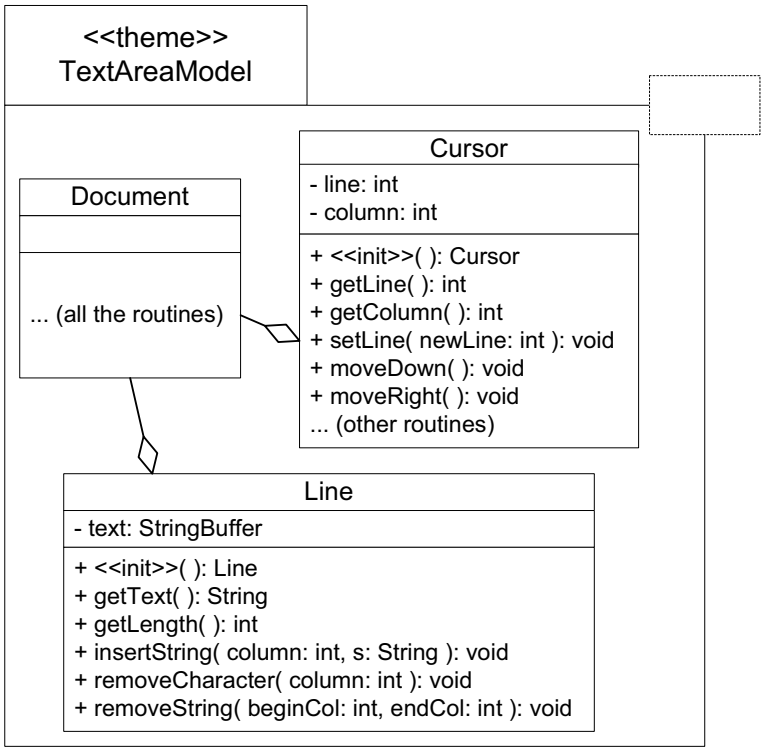


Figura 4.9: Diagrama de classes do tema **TextAreaModel**.

```
public class Speaker{
    public Speaker( ){
        this.listeners = new HashSet<Listener>( );
    }

    public void addListener( Listener l ){
        this.listeners.add( l );
    }

    private void fire_eventHappened( ){
        for( Listener listener : this.listeners )
            listener.eventHappened( );
    }

    private Set<Listener> listeners;
}

public interface Listener{
    void eventHappened( );
}
```

Note que a classe *Speaker* do tema **Monitoring** possui uma rotina privada que não é chamada por nenhuma outra rotina naquele tema. Isso sinaliza que a rotina chamadora está em outro tema, no caso o **AbstractDocumentMonitoring**, cuja implementação aparece a seguir.

```
abstract public class Document{
    private void change( ){
        proceed;
        this.fire_documentChanged( );
    }

    abstract private void fire_documentChanged( );
}
```

A composição entre **AbstractDocumentMonitoring** e **Monitoring** gera o tema **DocumentMonitoring**. Antes da composição, contudo, as classes reutilizáveis de **Monitoring** devem passar por uma redefinição. A fórmula de composição e a implementação do tema **DocumentMonitoring** são mostrados a seguir.

```
DocumentMonitoring := AbstractDocumentMonitoring + Monitoring{
    Speaker => Document;
    Document.listeners => Document.docListeners;
    Listener => simplenotepad.event.DocumentListener;
    Document.fire_eventHappened( ) => fire_documentChanged( );
    simplenotepad.event.DocumentListener.eventHappened( ) => documentChanged( );
    Document.addListener( simplenotepad.event.DocumentListener ) =>
     addDocumentListener( simplenotepad.event.DocumentListener );
};
```

```
public class Document{
    public Document( ){
        this.docListeners = new HashSet<DocumentListener>( );
    }

    public void addDocumentListener( DocumentListener l ){
        this.docListeners.add( l );
    }

    private void change( ){
        proceed;
        this.fire_documentChanged( );
    }

    private void fire_documentChanged( ){
        for( DocumentListener listener : this.docListeners )
            listener.documentChanged( );
    }

    private Set<DocumentListener> docListeners;
}
```

A composição entre **DocumentMonitoring** e **TextAreaModel** resulta no tema **TextAreaObservableModel**, que contém a funcionalidade básica do modelo mais a funcionalidade necessária para permitir à visão o monitoramento do modelo. A seguir aparece a fórmula de composição que gera o tema **TextAreaObservableModel**, cuja diferença com o tema **TextAreaModel** é apenas a implementação da classe *Document*, que foi complementada com a implementação do monitoramento da mudança de estado.

```
TextAreaObservableModel := TextAreaModel + DocumentMonitoring{
    Document.change( ) => moveCursor*( ) from TextAreaModel:Document,
     offsetCursorColumn( int ), setCursorColumn( int newColumn ),
     insertString( String s ), removeCharacter( ), createLineBreak( ),
     appendNextToCurrentLine( );
};
```

Agora o tema **TextAreaMVC**, que implementa a funcionalidade completa da área de texto para a primeira iteração, pode ser gerado. O tema **TextAreaDefaultConstructor**, que aparece na fórmula de composição mas não foi explicado anteriormente, apenas adiciona à classe *SimpleTextArea* um construtor que não recebe parâmetros, por conveniência. É também mostrado a seguir os dois construtores da classe *SimpleTextArea* resultante. O primeiro construtor é originário de **TextAreaDefaultConstructor**, enquanto o segundo contém código originário tanto de **TextAreaView** quanto de **TextAreaController**.

```
TextAreaMVC := TextAreaController + TextAreaView +
 TextAreaObservableModel + TextAreaDefaultConstructor;
```

```
public class SimpleTextArea extends JPanel{
    public SimpleTextArea( ){
        this( new Document( ) );
    }

    public SimpleTextArea( Document doc ){
        this.doc = doc;

        super.setBackground( Color.WHITE );
        this.font = new Font( "Monospaced", Font.PLAIN, 12 );
        super.setFont( this.font );

        this.doc.addDocumentListener( new STADocumentListener( ) );

        this.addKeyListener( new RemoveSelectedTextAndSTAKeyListener( doc ) );
        Action createLineAction = new RemoveSelectedTextAndCreateLineAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "pressed ENTER" ),
         createLineAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( createLineAction.getValue( Action.NAME ),
         createLineAction );
        Action moveCursorDownAction = new MoveCursorDownAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "pressed DOWN" ),
         moveCursorDownAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( moveCursorDownAction.getValue( Action.NAME ),
         moveCursorDownAction );
        Action moveCursorLeftAction = new MoveCursorLeftAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "pressed LEFT" ),
         moveCursorLeftAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( moveCursorLeftAction.getValue( Action.NAME ),
         moveCursorLeftAction );

        ... //other actions and keystrokes

        this.setFocusable( true );
        this.addFocusListener( new FocusAdapter( ){} );
    }

    ...
}
```

Feito isso, já se pode efetuar a tão esperada composição que gera o tema *SimpleNotepadApp*, o protótipo resultante da primeira iteração. Uma imagem da tela do programa aparece na Figura 4.10.

```
SimpleNotepadApp := CompleteWindowSetup + TextAreaMVC;
```

4.4 Segunda Iteração

Nesta iteração, serão mostrados menos detalhes de implementação para não tornar a explicação enfadonha e também porque muitas das mesmas idéias foram bem detalhadas na primeira iteração. Na segunda iteração, os seguintes novos requisitos deveriam ser satisfeitos:

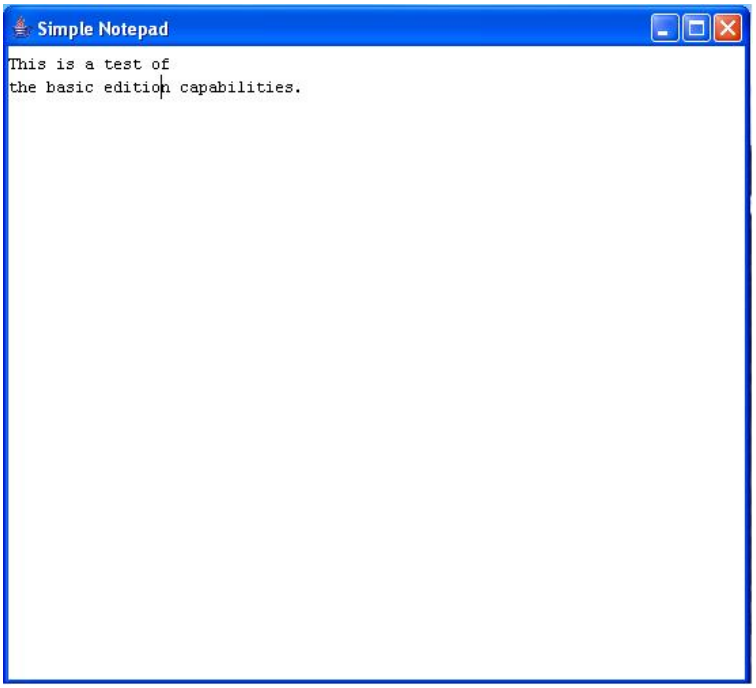


Figura 4.10: Imagem de tela do protótipo executável obtido na primeira iteração.

1. A estrutura da informação tratada pelo programa deve contemplar também uma posição auxiliar do cursor, chamada *mark*, para permitir seleção de trechos do documento. A posição do cursor que indica o ponto de edição é denominada *dot*.

2. As funcionalidades anteriores de entrada na área de texto devem ser mantidas, e mais algumas devem ser adicionadas:

   2 . 1 as teclas $shift$ e $HOME$ são pressionadas: o *dot* é movido para o início da linha atual, enquanto o *mark* se mantém;

   2 . 2 as teclas $shift$ e END são pressionadas: o *dot* é movido para o fim da linha atual, enquanto o *mark* se mantém;

   2 . 3 as teclas $shift$ e $\uparrow$ são pressionadas: o *dot* é movido para cima, enquanto o *mark* se mantém;

   2 . 4 as teclas $shift$ e $\downarrow$ são pressionadas: o *dot* é movido para baixo, enquanto o *mark* se mantém;

   2 . 5 as teclas $shift$ e $\leftarrow$ são pressionadas: o *dot* é movido para a esquerda, enquanto o *mark* se mantém;

   2 . 6 as teclas $shift$ e $\rightarrow$ são pressionadas: o *dot* é movido para a direita, enquanto o *mark* se mantém;

   2 . 7 qualquer caractere imprimível é digitado: todo o texto selecionado é removido e em seguida se efetua o que era feito no protótipo resultante da iteração anterior;

2 . 8 a tecla $ENTER$ é pressionada: todo o texto selecionado é removido e em seguida se efetua o que era feito no protótipo resultante da iteração anterior;

2 . 9 a tecla $DELETE$ é pressionada: se houver algum texto selecionado, todo o texto selecionado é removido, senão se efetua o que era feito no protótipo resultante da iteração anterior;

2 . 10 a tecla $BACK\_SPACE$ é pressionada: se houver algum texto selecionado, todo o texto selecionado é removido, senão se efetua o que era feito no protótipo resultante da iteração anterior;

3. Deve ser fornecido retorno adequado ao usuário caso um fragmento do documento esteja selecionado.

É importante notar que os requisitos selecionados para esta iteração seriam diferentes caso o critério de priorização adotado fosse o de maior valor para o usuário primeiro. Naquele critério, provavelmente seriam escolhidos requisitos relacionados a salvar os documentos em arquivos e a carregar documentos de arquivos, já que estas idéias são fundamentais a um sistema de edição de textos: ninguém usaria tal programa para editar um documento e logo em seguida descartá-lo, a não ser nos casos excepcionais em que o usuário não gostou do documento criado ou em que o usuário utilizou o programa como um recurso temporária de anotação de idéias. É claro que funcionalidades referentes à seleção de fragmentos do documento são importantes para a edição, mas não são tão importantes quanto salvar e carregar, considerando o propósito maior do sistema. Entretanto, o critério de priorização de requisitos aqui adotado levou em consideração apenas a conveniência do desenvolvedor.

Os requisitos indicam que devem ocorrer alterações em alguns dos principais temas do sistema *Simple Notepad*. O tema **TextAreaModel** deve ser alterado para acomodar o novo conceito de *mark* como posição auxiliar do cursor; o tema **TextAreaController** deve ser alterado para acomodar as novas funcionalidades de entrada; e o tema **TextAreaView** deve ser alterado para fornecer retorno ao usuário em caso de seleção, destacando o texto selecionado.

Poder-se-ia pensar que essas alterações indicam que a arquitetura do programa é frágil, pois um programa com uma arquitetura adequada não sofreria alterações em tantos elementos. Na verdade, observa-se o contrário: a arquitetura é resistente a mudanças porque somente alguns elementos devem ser individualmente alterados em face de requisitos não-antecipados; a arquitetura em si se mantém intacta. Frequentemente não é possível antecipar todos os requisitos que devem ser satisfeitos em iterações futuras, e também não é viável criar um programa tão flexível a ponto de acomodar possíveis requisitos futuros, dos quais não há certeza quanto a sua real solicitação. Tal flexibilidade viria acompanhada de complexidade adicional que tornaria difícil a evolução de software para requisitos que não puderam ser antecipados. Sendo assim, é melhor implementar o software da maneira mais simples possível, obviamente utilizando padrões arquiteturais relativamente estáveis para tornar menos traumática a evolução, que em geral terá de ocorrer de qualquer maneira.

Para esta iteração, o modelo de estrutura da informação foi concebido conforme exposto na Figura 4.11. A única mudança perceptível foi a inserção de duas novas posições no Cursor: *markLine* e *markColumn*, que juntas compõem o *mark* do cursor.

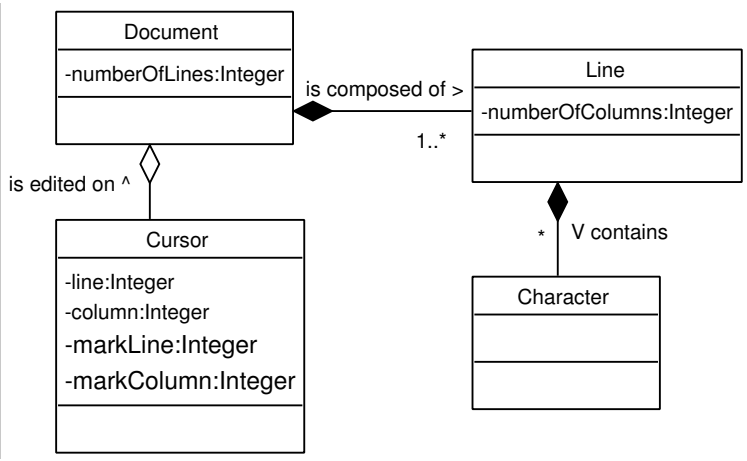


Figura 4.11: Modelo de estrutura da informação do *Simple Notepad* para a segunda iteração. Perceba as novas propriedades do cursor.

4.4.1 Tema TextAreaController

Uma das alterações a serem feitas envolve o tema **TextAreaController**. Contudo, apesar das alterações, é desejável manter o nome desse tema, pois ele ainda conserva o mesmo papel na arquitetura da área de texto. Assim, a primeira refatoração a ser feita é conservar o nome **TextAreaController** para indicar o controlador completo e renomear o controlador antigo, que agora será apenas parte do controlador completo.

```
TextAreaBasicController := ...; //antigo TextAreaController
TextAreaController := TextAreaBasicController; /* por enquanto o antigo controlador
 é tudo o que temos */
```

É interessante colocar as novas funcionalidades sob um único tema, denominado **TextAreaSelectionController**, porque todas as funcionalidades escolhidas envolvem seleção, que complementa as funcionalidades anteriormente disponibilizadas. Assim fazendo, minimiza-se a necessidade de mudanças invasivas na implementação dos temas anteriores.

```
TextAreaBasicController := ...; //antigo TextAreaController
TextAreaSelectionController := ...; //novas funcionalidades de entrada
TextAreaController := TextAreaBasicController + TextAreaSelectionController;
```

Para satisfazer os requisitos 2.1 a 2.6, foram criados os temas **MoveCursorDotToHome**, **MoveCursorDotToEnd**, **MoveCursorDotUp**, **MoveCursorDotDown**, **MoveCursorDotLeft** e **MoveCursorDotRight**, todos com estruturas bastante parecidas entre si e parecidas com a maioria

dos temas que compunham **TextAreaController** na primeira iteração. Com a finalidade ilustrativa, partes da implementação do tema **MoveCursorDotToHome** são mostradas a seguir.

```
abstract public class SimpleTextArea{
    public SimpleTextArea( Document doc ){
        Action moveCursorDotToHomeAction = new MoveCursorDotToHomeAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "shift pressed HOME" ),
         moveCursorDotToHomeAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( moveCursorDotToHomeAction.getValue( Action.NAME ),
         moveCursorDotToHomeAction );
    }

    abstract private InputMap getInputMap( );

    abstract private ActionMap getActionMap( );
}
```

```
public class MoveCursorDotToHomeAction extends AbstractAction{
    public MoveCursorDotToHomeAction( Document doc )
     throws NullPointerException{
        super( "Move cursor dot to home" );

        if( doc == null )
            throw new NullPointerException( );

        this.doc = doc;
        this.updateEnabling( );
    }

    public void actionPerformed( ActionEvent e )
     throws IllegalStateException{
        if( !this.isEnabled( ) )
            throw new IllegalStateException( );

        this.doc.moveCursorDotToHome( );
    }

    public void setEnabled( boolean enabled )
     throws UnsupportedOperationException{
        throw new UnsupportedOperationException( );
    }

    private void updateEnabling( ){
        super.setEnabled( true );
    }

    private Document doc;
}
```

```
abstract public class Document{
    abstract public void moveCursorDotToHome( );
}
```

Os requisitos 2.7 a 2.10 são interessantes porque requerem alterações nas funcionalidades anteriores do programa, e não simplesmente novas funcionalidades. Mesmo nesses casos, é possível

criar temas primitivos que não exigem alteração da implementação interna dos temas primitivos da iteração anterior.

Para satisfazer os requisitos 2.7 e 2.8, foi criado o tema **RemoveSelectedTextAndProceed**, que altera a semântica de alguns temas anteriores para remover o texto selecionado primeiro e só então efetuar o comportamento que ocorria anteriormente. A essência da implementação de **RemoveSelectedTextAndProceed** aparece a seguir.

```
abstract public class RemoveSelectedTextAndProceedAction{
    public void removeSelectedTextAndProceed( ){
        this.doc.removeSelectedText( );
        proceed;
    }

    private Document doc;
}
```

```
abstract public class Document{
    abstract public void removeSelectedText( );
}
```

Observe a existência de uma rotina incompleta, *removeSelectedTextAndProceed( )* na classe *RemoveSelectedTextAndProceedAction*. Essa rotina contém a implementação essencial do tema **RemoveSelectedTextAndProceed**, que indica a remoção do texto selecionado antes de se efetuar qualquer outra ação. A combinação desse fragmento de implementação com outros fragmentos será demonstrada mais adiante nesta seção.

Os requisitos 2.9 e 2.10 são satisfeitos pelo tema **RemoveSelectedTextOrProceed**, que altera a semântica de alguns temas anteriores para ou remover o texto selecionado ou efetuar a ação que ocorria anteriormente. A essência da implementação de **RemoveSelectedTextOrProceed** aparece a seguir, na qual as rotinas *removeSelectedTextOrProceed( )* e *updateEnabling( )* da classe *RemoveSelectedTextOrProceedAction* são apresentadas como incompletas.

```
public class RemoveSelectedTextOrProceedAction{
    public void removeSelectedTextOrProceed( ){
        if( this.doc.isTextSelected( ) )
            this.doc.removeSelectedText( );
        else
            proceed;
    }

    private void updateEnabling( ){
        proceed;
        super.setEnabled( this.isEnabled( ) || this.doc.isTextSelected( ) );
    }

    private Document doc;
}
```

```
abstract public class Document{
    abstract public boolean isTextSelected( );

    abstract public void removeSelectedText( );
}
```

Parte da implementação do tema **TextAreaSelectionController**, que resulta da composição dos temas mencionados anteriormente, é mostrada a seguir. Como esperado, não há nenhum fragmento de implementação referente aos temas **RemoveSelectedTextAndProceed** e **RemoveSelectedTextOrProceed** no construtor da classe *SimpleTextArea*.

```
TextAreaSelectionController := MoveCursorDotDown + MoveCursorDotLeft
+ MoveCursorDotRight + MoveCursorDotToEnd + MoveCursorDotToHome +
MoveCursorDotUp + RemoveSelectedTextOrProceed +
RemoveSelectedTextAndProceed;
```

```
public class SimpleTextArea{
    public SimpleTextArea( Document doc ){
        Action moveCursorDotDownAction = new MoveCursorDotDownAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "shift pressed DOWN" ),
         moveCursorDotDownAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( moveCursorDotDownAction.getValue( Action.NAME ),
         moveCursorDotDownAction );
        Action moveCursorDotLeftAction = new MoveCursorDotLeftAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "shift pressed LEFT" ),
         moveCursorDotLeftAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( moveCursorDotLeftAction.getValue( Action.NAME ),
         moveCursorDotLeftAction );
        Action moveCursorDotRightAction = new MoveCursorDotRightAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "shift pressed RIGHT" ),
         moveCursorDotRightAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( moveCursorDotRightAction.getValue( Action.NAME ),
         moveCursorDotRightAction );
        Action moveCursorDotToEndAction = new MoveCursorDotToEndAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "shift pressed END" ),
         moveCursorDotToEndAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( moveCursorDotToEndAction.getValue( Action.NAME ),
         moveCursorDotToEndAction );
        Action moveCursorDotToHomeAction = new MoveCursorDotToHomeAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "shift pressed HOME" ),
         moveCursorDotToHomeAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( moveCursorDotToHomeAction.getValue( Action.NAME ),
         moveCursorDotToHomeAction );
        Action moveCursorDotUpAction = new MoveCursorDotUpAction( doc );
        this.getInputMap( ).put( KeyStroke.getKeyStroke( "shift pressed UP" ),
         moveCursorDotUpAction.getValue( Action.NAME ) );
        this.getActionMap( ).put( moveCursorDotUpAction.getValue( Action.NAME ),
         moveCursorDotUpAction );
    }

    ...
}
```

No novo **TextAreaController**, a principal alteração ocorrerá no construtor da classe *SimpleTextArea*, que será complementado com o construtor existente no tema **TextAreaSelectionController**. O tema **TextAreaController** é gerado conforme a fórmula de composição a seguir.

```
TextAreaBasicController := ...; //antigo TextAreaController

TextAreaSelectionController := MoveCursorDotDown + MoveCursorDotLeft
+ MoveCursorDotRight + MoveCursorDotToEnd + MoveCursorDotToHome +
MoveCursorDotUp + RemoveSelectedTextOrProceed +
RemoveSelectedTextAndProceed;

TextAreaController := TextAreaBasicController{
    STAKeyListener => RemoveSelectedTextAndSTAKeyListener;
    CreateLineAction => RemoveSelectedTextAndCreateLineAction;
    RemovePreviousCharacterOrAppendCurrentToPreviousLineAction =>
     RemoveSelectedTextOrRemovePreviousCharacterOrAppendCurrentToPreviousLineAction;
    RemoveNextCharacterOrAppendNextToCurrentLineAction =>
     RemoveSelectedTextOrRemoveNextCharacterOrAppendNextToCurrentLineAction;
} +
 TextAreaSelectionController{
    RemoveSelectedTextAndProceedAction => RemoveSelectedTextAndSTAKeyListener{
        removeSelectedTextAndProceed( ) => processInsertion( String s );
    }, RemoveSelectedTextAndCreateLineAction{
        removeSelectedTextAndProceed( ) => actionPerformed( ActionEvent e );
    };
    RemoveSelectedTextOrProceedAction =>
     RemoveSelectedTextOrRemovePreviousCharacterOrAppendCurrentToPreviousLineAction{
        removeSelectedTextOrProceed( ) => actionPerformed( ActionEvent e );
    }, RemoveSelectedTextOrRemoveNextCharacterOrAppendNextToCurrentLineAction{
        removeSelectedTextOrProceed( ) => actionPerformed( ActionEvent e );
    };
};
```

4.4.2 Tema TextAreaView

Com o tema **TextAreaView** ocorre algo parecido com o que ocorreu com o tema **TextAreaController**. Deve-se aumentar o tema **TextAreaView** para que este contenha implementação relativa ao destaque de texto selecionado. Primeiramente, a fórmula de composição é refatorada para guardar separadamente a funcionalidade de visão do programa anterior, agora parte da visão completa.

```
TextAreaBasicView := ...; //antiga TextAreaView
TextAreaView :=TextAreaBasicView; /* por enquanto a antiga visão é
tudo o que temos */
```

Como as funcionalidades que envolvem seleção apenas complementam as funcionalidades anteriores, é interessante colocar as novas funcionalidades sob um único tema, denominado **TextAreaSelectionView**.

```
TextAreaBasicView := ...; //antigo TextAreaView
TextAreaSelectionView := ...; //novas funcionalidades de saída
TextAreaView := TextAreaBasicView + TextAreaSelectionView;
```

A implementação de **TextAreaSelectionView** é formada pelas classes *SimpleTextArea* e *Document*, mostradas a seguir.

```
abstract public class SimpleTextArea extends JPanel{
    private void paintDocument( Graphics g ){
        this.paintSelection( g );
        proceed;
    }

    private void paintSelection( Graphics g ){
        int l1 = this.doc.getSelectionStartLine( );
        int l2 = this.doc.getSelectionEndLine( );
        int c1 = this.doc.getSelectionStartColumn( );
        int c2 = this.doc.getSelectionEndColumn( );

        while( l1 < l2 ){
            this.paintSelectionInLine( g, l1, c1,
             this.doc.getLineLength( l1 ) + 1 );
            l1 += 1;
            c1 = 1;
        }

        this.paintSelectionInLine( g, l1, c1, c2 );
    }

    private void paintSelectionInLine( Graphics g, int line, int startCol,
     int endCol ){
        int x1 = this.getX( line, startCol );
        int x2 = this.getX( line, endCol );
        int y1 = this.getY( line ) - super.getFontMetrics( font ).getAscent( );
        int y2 = this.getY( line ) + super.getFontMetrics( font ).getDescent( );

        g.setColor( Color.BLUE );
        g.fillRect( x1, y1, x2 - x1, y2 - y1 );
    }

    private void paintCharacter( Graphics g, int line, int column ){
        if( this.doc.isPointSelected( line, column ) )
            g.setColor( Color.WHITE );
        else
            g.setColor( Color.BLACK );

        proceed;
    }

    abstract private int getX( int line, int column );

    abstract private int getY( int line );

    private Document doc;
    private Font font;
}
```

```
abstract public class Document{
    abstract public int getLineLength( int line );

    abstract public int getSelectionStartLine( );

    abstract public int getSelectionStartColumn( );

    abstract public int getSelectionEndLine( );

    abstract public int getSelectionEndColumn( );

    abstract public boolean isPointSelected( int line, int column );
}
```

4.4.3 Tema TextAreaModel

Algumas rotinas utilizadas por **TextAreaSelectionController** e **TextAreaSelectionView** que não estavam implementadas no antigo tema **TextAreaModel** (agora renomeado para **TextAreaBasicModel**) devem ser agora implementadas. Em termos de rotinas públicas, adotou-se como critério de *design* que o novo tema **TextAreaSelectionModel** conteria todas e somente as rotinas que envolvem funcionalidades relacionadas à seleção, enquanto o tema **TextAreaBasicModel** conteria as demais rotinas. As rotinas privadas seriam colocadas aonde fossem necessárias. O compromisso foi de sacrificar a inconsciência dos temas se necessário para se obter maior coesão interna em cada tema. Isto significa que se modificaria se necessário a implementação original de **TextAreaBasicModel** para conter funcionalidades não-relacionadas à seleção que não eram necessárias na iteração anterior. Felizmente, para as circunstâncias desta iteração, não foi necessário sacrificar a inconsciência do tema **TextAreaBasicModel**, mas não se hesitaria em fazê-lo caso fosse necessário. Assim, na prática o tema **TextAreaSelectionModel** acabou contendo a implementação de todas as rotinas públicas utilizadas por **TextAreaSelectionController** e **TextAreaSelectionView** que não estavam implementadas no antigo tema **TextAreaModel** (que agora é **TextAreaBasicModel**). O diagrama de classes correspondente ao tema **TextAreaSelectionModel** é mostrado na Figura 4.12. Perceba que a classe *Cursor* possui agora dois novos atributos, correspondentes à linha e à coluna do *mark*.

O tema **TextAreaModel** desta iteração, composto por **TextAreaBasicModel** e **TextAreaSelectionModel**, mantém-se como o tema que contém a implementação de todas as rotinas utilizadas por **TextAreaController** e **TextAreaView**. A composição que gera **TextAreaModel** é mostrada a seguir.

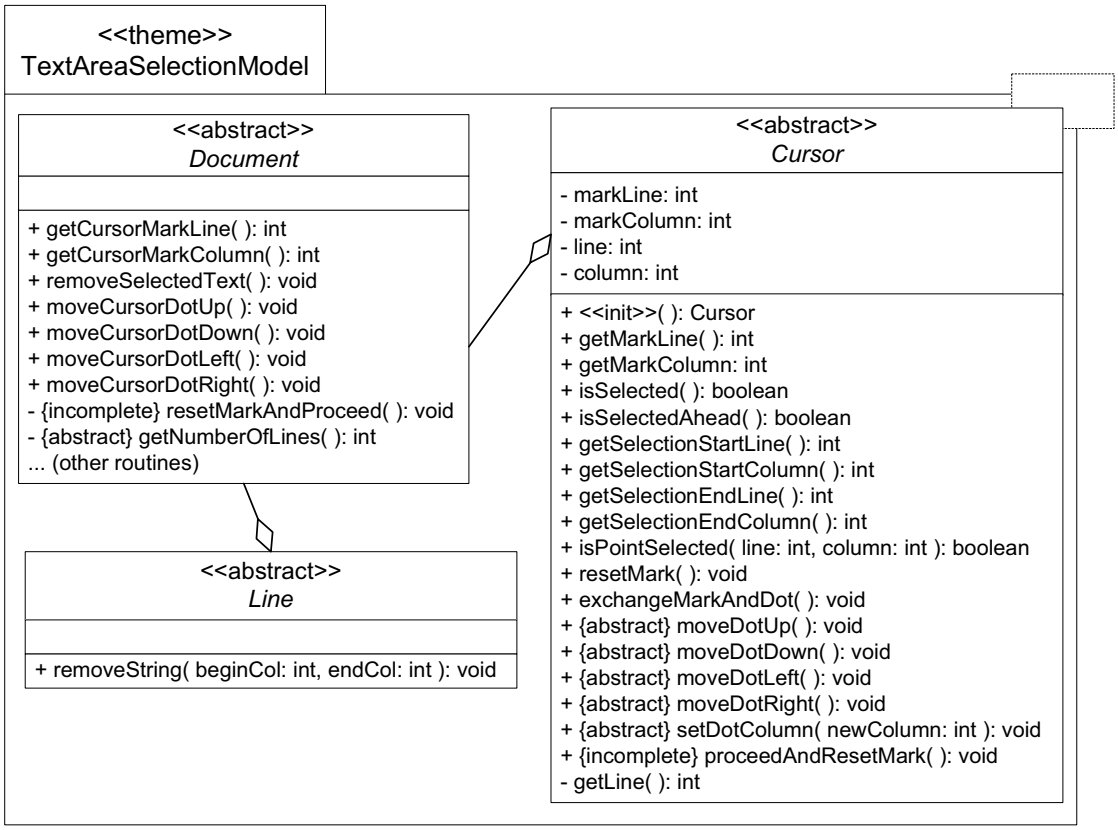


Figura 4.12: Diagrama de classes do tema **TextAreaSelectionModel**.

```
TextAreaBasicModel_extended := TextAreaBasicModel{
    Cursor.moveUp( ) => moveUp( ), moveDotUp( );
    Cursor.moveDown( ) => moveDown( ), moveDotDown( );
    Cursor.moveLeft( ) => moveLeft( ), moveDotLeft( );
    Cursor.moveRight( ) => moveRight( ), moveDotRight( );
    Cursor.setColumn( int ) => setColumn( int ), setDotColumn( int );
};

TextAreaModel := TextAreaBasicModel_extended +
TextAreaSelectionModel{
    Cursor.proceedAndResetMark( ) => move*( ) - moveDot*( ) from
     TextAreaBasicModel_extended:Cursor, setColumn( int newColumn ),
      offsetColumn( int offset ), setLine( int line );
    Document.resetMarkAndProceed( ) => createLineBreak( ); /* should
     also be converted to insertString( String s ), removeCharacter( )
     and appendNextToCurrentLine( ) */
};
```

O tema **TextAreaObservableModel** se mantém como o resultado da composição entre **TextAreaModel** e **DocumentMonitoring**. Contudo, algumas rotinas novas tiveram de ser redefinidas na fórmula de composição porque elas também alteram o estado do documento.

```
TextAreaObservableModel := TextAreaModel + DocumentMonitoring{
    Document.change( ) => moveCursor*( ) from TextAreaModel:Document,
     offsetCursorColumn( int ), setCursorColumn( int newColumn ),
     insertString( String s ), removeCharacter( ), createLineBreak( ),
     appendNextToCurrentLine( ),
     //new routines
     setCursorDotColumn( int newDotColumn ), removeSelectedText( );
};
```

As regras de composição que geram os temas **TextAreaMVC** e **SimpleNotepadApp** permanecem as mesmas da primeira iteração.

4.4.4 Problemas na redefinição da pré-condição de rotinas em *Java*

Em vista da inserção de funcionalidades de seleção, algumas rotinas da classe *Document* do tema **TextAreaBasicModel** necessitaram ter a semântica ligeiramente alterada. A partir de agora, as rotinas *createLineBreak( )*, *insertString(String)*, *removeCharacter( )* e *appendNextToCurrentLine( )* ajustam a linha e a coluna do *mark* para serem respectivamente iguais à linha e à coluna do *dot*. Isso foi feito para se evitar problemas de integridade do documento: o que aconteceria com *mark* e *dot*, por exemplo, se um caractere fosse removido e o *dot* aparecesse antes do *mark*?

Em princípio, para esta alteração semântica, seria necessário combinar a rotina *resetMarkAndProceed( )* da classe *Document* do tema **TextAreaSelectionModel** com as rotinas mencionadas. Contudo, ao fazê-lo, aparecem problemas referentes ao contrato de cada rotina. Observe a seguir as rotinas *insertString(String)* de **TextAreaBasicModel**, *resetMarkAndProceed( )* de **TextAreaSelectionModel**, e o resultado da combinação das duas rotinas.

```
//in TextAreaBasicModel
public class Document{
    public void insertString( String s )
     throws NullPointerException{
        //precondition code
        if( s == null )
            throw new NullPointerException( );

        //``real meat''
        this.currentLine( ).insertString( this.cursor.getColumn( ), s );
    }
}
```

```
//in TextAreaSelectionModel
abstract public class Document{
    private void resetMarkAndProceed( ){
        this.cursor.resetMark( );
        proceed;
    }
}
```

```
//composition result
public class Document{
    public void insertString( String s )
     throws NullPointerException{
        //resetMarkAndProceed code
       this.cursor.resetMark( );

        //precondition code
        if( s == null )
            throw new NullPointerException( );

        //``real meat''
        this.currentLine( ).insertString( this.cursor.getColumn( ), s );
    }
}
```

O problema é que o código da rotina *resetMarkAndProceed( )* aparece antes do código da *pré-condição* da rotina *insertString(String)*. Do ponto de vista do contrato das rotinas, o que deveria ocorrer nesse caso é que, quando as duas rotinas são combinadas, a pré-condição das duas rotinas aparece antes, e só depois do código "real" da rotina combinada aparece. A seguir é demonstrado de maneira genérica como ocorre e como deveria ocorrer.

```
//in Theme1
public class AClass{
    public void aRoutine( ){
        ... //precondition code from Theme1:AClass.aRoutine( )
        ... //``real meat'' from Theme1:AClass.aRoutine( )
    }
}
```

```
//in Theme2
public class AClass{
    public void aRoutine( ){
        ... //precondition code from Theme2:AClass.aRoutine( )
        ... //``real meat'' from Theme2:AClass.aRoutine( )
    }
}
```

```
//in the resulting theme (what is, unfortunately)
public class AClass{
    public void aRoutine( ){
        ... //precondition code from Theme1:AClass.aRoutine( )
        ... //``real meat'' from Theme1:AClass.aRoutine( )
        ... //precondition code from Theme2:AClass.aRoutine( )
        ... //``real meat'' from Theme2:AClass.aRoutine( )
    }
}
```

```
//in the resulting theme (what should be)
public class AClass{
    public void aRoutine( ){
        ... //precondition code from Theme1:AClass.aRoutine( )
        ... //precondition code from Theme2:AClass.aRoutine( )
        ... //``real meat'' from Theme1:AClass.aRoutine( )
        ... //``real meat'' from Theme2:AClass.aRoutine( )
    }
}
```

Na implementação baseada em *Java*, o código "real" da primeira rotina aparece antes de se verificar se a pré-condição da segunda rotina é satisfeita. O risco é que a rotina combinada pode ser chamada em um contexto que satisfaz a pré-condição da primeira rotina mas não satisfaz a pré-condição da segunda rotina. Neste caso, o código da primeira rotina seria executado, com todos os seus efeitos colaterais, e em seguida a rotina combinada iria falhar, possivelmente gerando uma exceção após se verificar que a pré-condição da segunda rotina não é satisfeita.

Este problema só ocorre porque a linguagem de programação *Java* não possui a capacidade de separar explicitamente o código da pré-condição do código "real" da rotina. Embora não se tenha experimentado, é possível que uma abordagem que utiliza a linguagem *Eiffel* como substrato não sofreria deste problema, pois *Eiffel* separa a pré-condição da implementação da rotina em si.

Uma solução possível é duplicar a pré-condição da segunda rotina na primeira rotina, para que ela seja realmente executada antes de qualquer efeito colateral ocorrer, como mostrado a seguir.

```
//in Theme1
public class AClass{
    public void aRoutine( ){
        ... //precondition code from Theme1:AClass.aRoutine( )
        ... //precondition code from Theme2:AClass.aRoutine( )
        ... //``real meat'' from Theme1:AClass.aRoutine( )
    }
}
```

```
//in Theme2
public class AClass{
    public void aRoutine( ){
        ... //precondition code from Theme2:AClass.aRoutine( )
        ... //``real meat'' from Theme2:AClass.aRoutine( )
    }
}
```

```
//in the resulting theme (what is, unfortunately)
public class AClass{
    public void aRoutine( ){
        ... //precondition code from Theme1:AClass.aRoutine( )
        ... //precondition code from Theme2:AClass.aRoutine( )
        ... //``real meat'' from Theme1:AClass.aRoutine( )
        ... //precondition code from Theme2:AClass.aRoutine( )
        ... //``real meat'' from Theme2:AClass.aRoutine( )
    }
}
```

Contudo, esta solução não é plenamente satisfatória por dois motivos. Primeiramente, a execução da verificação da pré-condição da segunda rotina é feita duas vezes, o que pode acarretar queda no desempenho do programa. Além disso, pode ser que o efeito colateral da implementação da primeira rotina coloque o objeto em um estado que viola a pré-condição da segunda rotina, que não era violada anteriormente, daí gerando uma exceção que não poderia ocorrer, o que fere a semântica da rotina. Portanto, esta solução só pode ser utilizada quando o efeito colateral da implementação da primeira rotina não viola a pré-condição da segunda rotina e a queda do desempenho é negligível. O autor desta monografia não encontrou uma solução mais satisfatória quando se utiliza *Java* como substrato. É provável que a única solução realmente satisfatória seja utilizar uma linguagem que separe o código de verificação da pré-condição da implementação real da rotina.

Esta solução foi utilizada no estudo de caso para complementar as rotinas *insertString(String)*, *removeCharacter( )* e *appendNextToCurrentLine( )* da classe *Document* do tema *TextAreaBasicModel* com código que torna o *mark* do cursor igual ao *dot*. As rotinas que as complementam possuem os mesmos nomes, e estão presentes na classe *Document* do tema *TextAreaSelectionModel*. Somente a rotina *createLineBreak( )* pôde ser composta com *resetMarkAndProceed( )* porque sua pré-condição é vazia ou verdadeira, isto é, a rotina pode ser chamada em qualquer circunstância. As demais rotinas da classe *Document* não necessitaram ter a semântica alterada.

Uma imagem de tela do programa obtido nesta iteração aparece na Figura 4.13.

4.5 Terceira Iteração

As características relativas ao controlador da área de texto que foram adicionadas até agora são de *interação direta* entre o usuário humano e o *widget*. Nesta iteração, ainda considerando somente a conveniência para o desenvolvedor, serão adicionadas algumas características de *interação indireta*, pelas quais o usuário não interage diretamente com o *widget*-alvo para requisitar a execução da funcionalidade, mas com outro mecanismo que serve de intermediário. No caso, o usuário interagirá com itens de menu, cujos objetos de ação irão chamar rotinas disponibilizadas pela interface de programação da área de texto. Os requisitos para esta iteração aparecem a seguir.

As seguintes funcionalidades que agem sobre a área de texto devem estar disponíveis por meio de itens de menu:

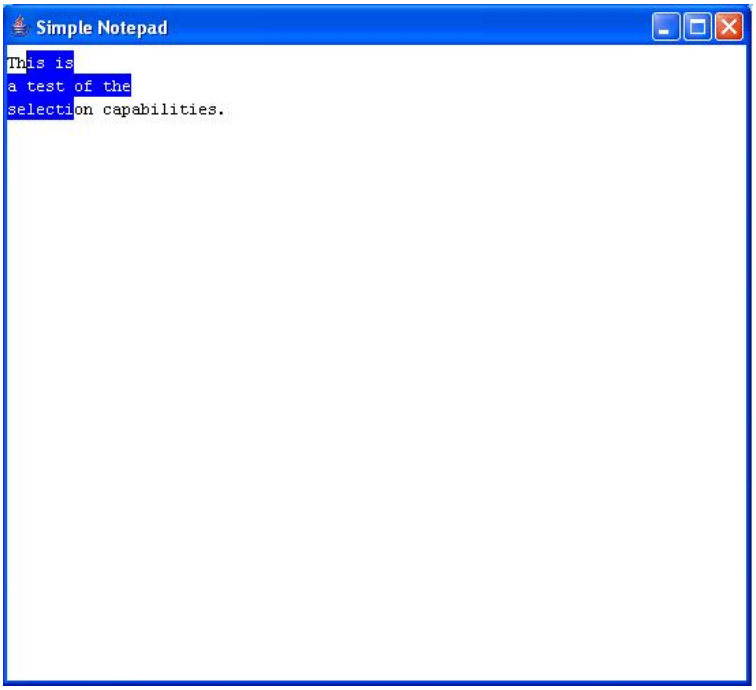


Figura 4.13: Imagem de tela do protótipo executável obtido na segunda iteração.

1. Criar um novo documento;
2. Sair do sistema;
3. Copiar o fragmento de documento selecionado, possivelmente composto por múltiplas linhas, do documento para uma área de transferência;
4. Recortar o fragmento de documento selecionado, possivelmente composto por múltiplas linhas, do documento para uma área de transferência;
5. Colar o conteúdo da área de transferência, possivelmente composto por múltiplas linhas, no documento, conforme a posição atual do cursor, removendo um eventual fragmento de documento selecionado no momento da requisição de colar.

Perceba que o modelo do software agora deve incorporar o conceito de área de transferência. O novo modelo de estrutura da informação aparece na Figura 4.14. Além disso, na versão anterior do sistema não havia menus, itens de menu e barra de menus, elementos que agora tornam-se relevantes para a interface do sistema com o usuário.

A idéia de controlador de interação indireta implica que a área de texto não será a única responsável por processar diretamente a entrada do usuário. Como itens de menu serão ativados pelo usuário para afetar a área de texto, a interface do módulo da área de texto (classe *SimpleTextArea*) deverá ser complementada com novas rotinas que irão processar indiretamente a entrada que o usuário fez via itens de menu.

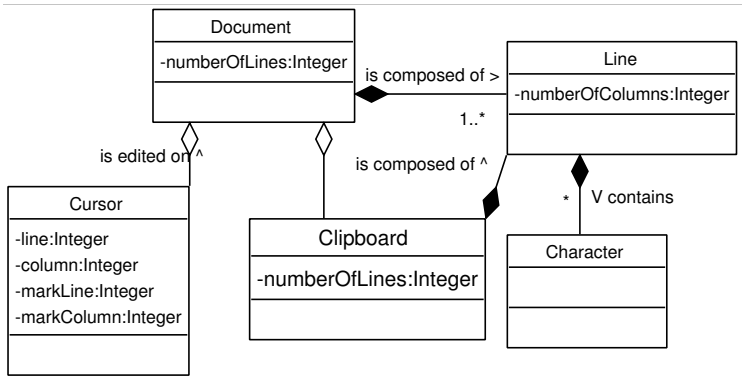


Figura 4.14: Modelo de estrutura da informação do *Simple Notepad* para a segunda iteração. Note a existência do novo conceito de área de transferência (*clipboard*).

Para a interface com o usuário, convencionou-se que a organização de todos os menus e itens de menu, independentemente de suas correspondentes funcionalidades, estaria contida em um único tema, denominado **CompleteMenuBarSetup**. Esse tema pode ser formado pela composição de outros temas menores, que conteriam grupos específicos de menus e de itens de menu, além da própria barra de menus.

Nas iterações anteriores, o programa completo era gerado pela seguinte fórmula de composição.

```
SimpleNotepadApp := CompleteWindowSetup + TextAreaMVC;
```

O tema **CompleteWindowSetup** continha apenas a implementação necessária para configurar a janela do programa, instanciar e inserir a área de texto na janela, e fechar o programa. Como agora a janela deve conter também uma barra de menus, o tema **CompleteWindowSetup** anterior torna-se apenas parte do que é necessário. Além disso, a implementação necessária para fechar o programa deve estar contida em um tema separado (denominado **BasicApplicationExitPolicy**) porque agora há duas maneiras de fechar o programa: pela janela (a maneira antiga) e por um item de menu, conforme o requisito 2. Assim, as seguintes alterações são feitas para gerar o programa completo. As redefinições necessárias são ignoradas no conjunto de fórmulas de composição a seguir.

```
BasicWindowSetup := ... //antigo CompleteWindowSetup

CompleteWindowSetup := BasicWindowSetup + CompleteMenuBarSetup +
 BasicApplicationExitPolicy;

SimpleNotepadApp := CompleteWindowSetup + TextAreaMVC;
```

4.5.1 Tema CompleteMenuBarSetup

Anteriormente, o tema **BasicWindowSetup** continha apenas a implementação para instanciar a janela e a área de texto, configurar a área de texto dentro da janela, e ajustar tamanho e visibilidade dos componentes. A antiga implementação da classe *SimpleNotepad* é mostrada a seguir.

O tema **CompleteMenuBarSetup** contém a implementação composta das hierarquias de menus e itens de menu na barra de menus (tema **MenuBarAssembly**), e também a implementação da inserção da barra de menus na janela principal do programa (tema **EmptyMenuBarInsertion**). O próprio tema **MenuBarAssembly** é gerado por meio da composição de temas que geram cada uma das hierarquias de menus.

```
CompleteFileMenuInsertion := FileMenuInsertion + NewMenuItemInsertionInFileMenu +
 ExitMenuItemInsertionInFileMenu;

CompleteEditMenuInsertion := EditMenuInsertion + CutMenuItemInsertionInEditMenu +
 CopyMenuItemInsertionInEditMenu + PasteMenuItemInsertionInEditMenu;

MenuBarAssembly := CompleteFileMenuInsertion + CompleteEditMenuInsertion;

CompleteMenuBarSetup := EmptyMenuBarInsertion + MenuBarAssembly;
```

Apenas para ilustrar a idéia, segue-se a implementação do tema **PasteMenuItemInsertionInEditMenu**. Os demais temas que envolvem itens de menu (que aparecem na forma ***XXX* MenuItemInsertionIn*YYY*Menu**) são similares.

```
public class SimpleNotepad{
    private void initWidgets( ){
        this.pasteItem = new JMenuItem( new PasteAction( this.area ) );
    }

    private void assemblyMenuBar( ){
        this.editMenu.add( this.pasteItem );
    }

    private SimpleTextArea area;
    private JMenu editMenu;
    private JMenuItem pasteItem;
}
```

```
abstract public class SimpleTextArea{
    abstract public boolean isClipboardEmpty( );

    abstract public void removeSelectedText( );

    abstract public void paste( );

    abstract public void addClipboardListener( ClipboardListener l );
}
```

```
public class PasteAction extends AbstractAction{
    public PasteAction( SimpleTextArea area )
     throws NullPointerException{
        super( "Paste" );

        if( area == null )
            throw new NullPointerException( );

        this.area = area;
        this.updateEnabling( );
        this.area.addClipboardListener( new PasteClipboardListener( ) );
    }

    public void actionPerformed( ActionEvent e )
     throws IllegalStateException{
        if( !this.isEnabled( ) )
            throw new IllegalStateException( );

        this.area.removeSelectedText( );
        this.area.paste( );
    }

    private void updateEnabling( ){
        this.setEnabled( !this.area.isClipboardEmpty( ) );
    }

    private SimpleTextArea area;


    private class PasteClipboardListener implements ClipboardListener{
        public void clipboardChanged( ){
            PasteAction.this.updateEnabling( );
        }
    }
}
```

Note que os novos temas requerem que algumas rotinas, que não existiam anteriormente, sejam disponibilizadas na classe *SimpleTextArea*. Isso inclui rotinas para acessar o estado, alterar o estado, e efetuar registro para monitorar ou observar alterações de estado da área de texto (em particular, a respeito da área de transferência associada com a área de texto).

Os temas na forma ***YYY*MenuInsertion** são similares. Para efeito de ilustração, é demonstrada a seguir a implementação de **EditMenuInsertion**.

```
public class SimpleNotepad{
    private void initWidgets( ){
        this.editMenu = new JMenu( "Edit" );
    }

    private void assemblyMenuBar( ){
        this.bar.add( this.editMenu );
    }

    private JMenuBar bar;
    private JMenu editMenu;
}
```

Perceba que os menus são inseridos em uma barra de menus. A instanciação e inserção desta barra de menus na janela é feita no tema **EmptyMenuBarInsertion**.

```
public class SimpleNotepad{
    public SimpleNotepad( ){
        this.assemblyMenuBar( );
    }

    private void initWidgets( ){
        this.bar = new JMenuBar( );
        this.window.setJMenuBar( bar );
    }

    private void assemblyMenuBar( ){
    }

    private JFrame window;
    private JMenuBar bar;
}
```

4.5.2 Tema BasicApplicationExitPolicy e mudança invasiva no tema BasicWindowSetup

Algumas mudanças invasivas foram necessárias no tema **BasicWindowSetup** (antigo **CompleteWindowSetup** para que se pudesse prover as novas funcionalidades requeridas nesta iteração. A abordagem de DSOT não impediu a violação da inconsciência do módulo neste caso. A implementação anterior da classe *ApplicationExitWindowListener* do tema **BasicWindowSetup** é exibida a seguir.

```
public class ApplicationExitWindowListener extends WindowAdapter{
    public void windowClosing( WindowEvent e ){
        this.execute( e.getWindow( ) );
    }

    private void execute( Window window ){
            window.setVisible( false );
            window.dispose( );
            System.exit( 0 );
    }
}
```

Tal implementação era suficiente para o propósito anterior de fechar o programa por meio da janela somente. Contudo, quando se insere o item de menu "Exit", o código da rotina *execute(Window)* acaba duplicado, conforme a implementação da classe *ApplicationExitAction* do tema **ExitMenuItemInsertionInFileMenu**.

```
public class ApplicationExitAction extends AbstractAction{
    public ApplicationExitAction( Window window ){
        super( "Exit" );

        this.window = window;
        this.updateEnabling( );
    }

    public void actionPerformed( ActionEvent e )
     throws IllegalStateException{
        if( !this.isEnabled( ) )
            throw new IllegalStateException( );

        this.execute( this.window );
    }

    public void setEnabled( boolean enabled )
     throws UnsupportedOperationException{
        throw new UnsupportedOperationException( );
    }

    private void updateEnabling( ){
        super.setEnabled( true );
    }

    /* code replication: see BasicWindowSetup theme */
    private void execute( Window window ){
        window.setVisible( false );
        window.dispose( );
        System.exit( 0 );
    }

    private Window window;
}
```

Esta duplicação de código é indesejada, já que caso a política de fechamento do programa seja alterada, dois locais terão de ser alterados, o que traz o risco de inconsistência. Portanto, é importante localizar tal implementação em apenas um módulo. Neste caso, o tema **BasicApplicationEx-**

itPolicy, mostrado a seguir, foi criado para guardar a implementação da rotina *execute(Window)*, enquanto as rotinas *execute(Window)* antigas foram tornadas abstratas.

```
public class ApplicationExitPolicy{
    private void execute( Window window ){
        window.setVisible( false );
        window.dispose( );
        System.exit( 0 );
    }
}
```

Como a classe *ApplicationExitPolicy* do tema **BasicApplicationExitPolicy** deve ser composta com a classe *ApplicationExitWindowListener* do tema **BasicWindowSetup** e com a classe *ApplicationExitAction* do tema **ExitMenuItemInsertionInFileMenu**, uma listagem de exceção deve aparecer na fórmula de composição para tratar estes casos que divergem do *default*.

```
CompleteWindowSetup := BasicWindowSetup + CompleteMenuBarSetup +
 BasicApplicationExitPolicy{
    ApplicationExitPolicy => ApplicationExitAction,
     ApplicationExitWindowListener;
}; /* CompleteMenuBarSetup resulta da composição de ExitMenuItemInsertionInFileMenu
 com outros temas */
```

4.5.3 Tema TextAreaMVC e as novas características de controlador indireto

Na discussão sobre o tema **MenuBarAssembly** e também no início da Seção 4.5, comentou-se que a classe *SimpleTextArea* teria de ser complementada com novas rotinas para poder sofrer a influência de itens de menu, mecanismos controladores de interação indireta. Isso significa que o tema **TextAreaMVC** terá de ser complementado para se adequar ao novo contexto. A implementação de **TextAreaMVC** segue.

```
TextAreaObservableModel := ...

TextAreaView := ...

TextAreaController := ...

TextAreaMVC := TextAreaController + TextAreaView +
 TextAreaObservableModel + TextAreaDefaultConstructor;
```

Perceba que a complementação requerida afeta somente a classe *SimpleTextArea*. Isso significa que o tema **TextAreaObservableModel** não terá de ser complementado por enquanto, pois ele não trata dessa classe. Então sobram os temas **TextAreaView**, **TextAreaController** e **TextAreaDefaultConstructor**. O tema **TextAreaView** não será complementado porque as alterações requeridas não envolvem o componente de visão da área de texto. O tema **TextAreaDefaultConstructor**

também não é alvo da complementação porque as alterações requeridas não envolvem alterações na chamada do construtor *default*. Portanto, resta o tema **TextAreaController**.

```
TextAreaController := TextAreaBasicController{
    STAKeyListener => RemoveSelectedTextAndSTAKeyListener;
    CreateLineAction => RemoveSelectedTextAndCreateLineAction;
    RemovePreviousCharacterOrAppendCurrentToPreviousLineAction =>
     RemoveSelectedTextOrRemovePreviousCharacterOrAppendCurrentToPreviousLineAction;
    RemoveNextCharacterOrAppendNextToCurrentLineAction =>
     RemoveSelectedTextOrRemoveNextCharacterOrAppendNextToCurrentLineAction;
} +
 TextAreaSelectionController{
    RemoveSelectedTextAndProceedAction => RemoveSelectedTextAndSTAKeyListener{
        removeSelectedTextAndProceed( ) => processInsertion( String s );
    }, RemoveSelectedTextAndCreateLineAction{
        removeSelectedTextAndProceed( ) => actionPerformed( ActionEvent e );
    };
    RemoveSelectedTextOrProceedAction =>
     RemoveSelectedTextOrRemovePreviousCharacterOrAppendCurrentToPreviousLineAction{
        removeSelectedTextOrProceed( ) => actionPerformed( ActionEvent e );
    }, RemoveSelectedTextOrRemoveNextCharacterOrAppendNextToCurrentLineAction{
        removeSelectedTextOrProceed( ) => actionPerformed( ActionEvent e );
    };
};
```

TextAreaController é o tema a ser afetado, pois seus subtemas, **TextAreaBasicController** e **TextAreaSelectionController**, contêm a implementação de características relativas a um controlador direto. As novas características requeridas, de controlador indireto, estarão implementadas em um novo tema, denominado **TextAreaIndirectController**. Esse tema contém a implementação de *SimpleTextArea* executada após ações sobre os itens de menu “Cut”, “Copy”, “Paste” e “New” terem sido processadas. A única alteração na fórmula de composição que gera **TextAreaController** é a adição de **TextAreaIndirectController**. Esse tema também não é primitivo, mas composto, conforme mostrado a seguir.

```
TextAreaIndirectController := TextAreaClearIndirectController +
 TextAreaClipboardIndirectController;
```

O tema **TextAreaClearIndirectController** contém a implementação da rotina *clear( )* de *SimpleTextArea*, que constitui o núcleo do processamento após o item de menu “New” ter sido acionado. Já o tema **TextAreaClipboardIndirectController** contém a implementação das várias rotinas de *SimpleTextArea* que cuidam do processamento de ações sobre a área de transferência (correspondentes às ações engatilhadas pelos itens de menu “Cut”, “Copy” e “Paste”). O diagrama de classes correspondente ao tema **TextAreaClipboardIndirectController** é apresentado na Figura 4.15.

Note que agora a classe *Document* deve possuir a implementação de várias rotinas novas, entre elas uma rotina para registrar um observador para mudanças de estado na área de transferência. Assim, o tema **TextAreaObservableModel** deve ser complementado.

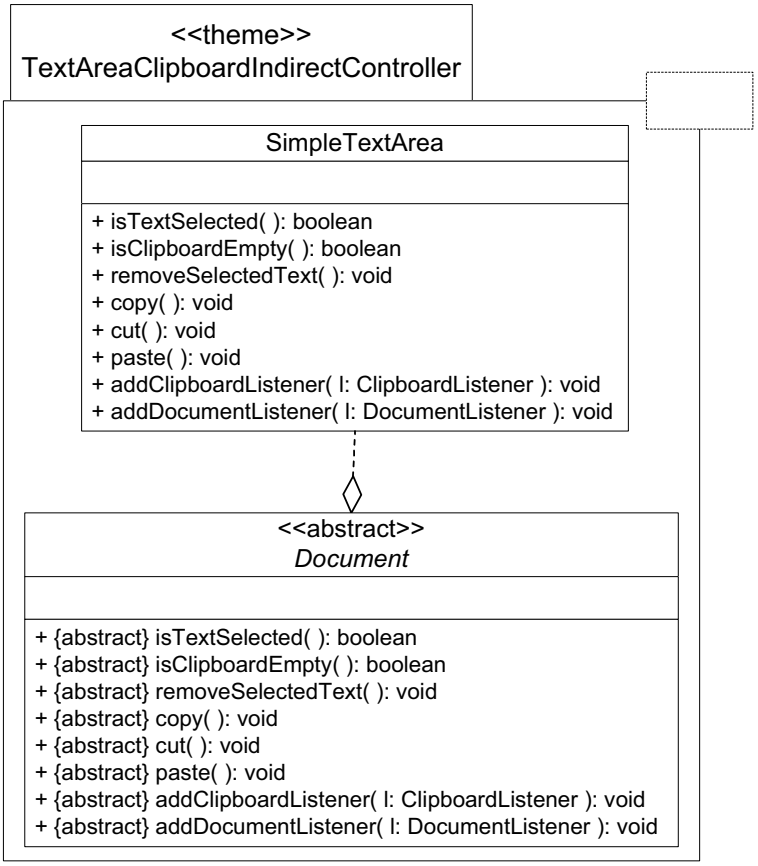


Figura 4.15: Diagrama de classes do tema **TextAreaClipboardIndirectController**. A classe *SimpleTextArea* apenas delega as chamadas para a classe *Document*.

4.5.4 Tema TextAreaObservableModel e a implementação da área de transferência no modelo

Conforme comentado na seção 4.5.3, a classe *SimpleTextArea* do tema **TextAreaIndirectController** requer a disponibilidade de algumas novas rotinas na classe *Document*, que deverão em última instância ser disponibilizadas no tema **TextAreaObservableModel**. Observe a seguir as fórmulas de composição que anteriormente originavam o referido tema.

```
TextAreaBasicModel_extended := TextAreaBasicModel{
    Cursor.moveUp( ) => moveUp( ), moveDotUp( );
    Cursor.moveDown( ) => moveDown( ), moveDotDown( );
    Cursor.moveLeft( ) => moveLeft( ), moveDotLeft( );
    Cursor.moveRight( ) => moveRight( ), moveDotRight( );
    Cursor.setColumn( int ) => setColumn( int ), setDotColumn( int );
};

TextAreaModel := TextAreaBasicModel_extended + TextAreaSelectionModel{
    Cursor.proceedAndResetMark( ) => move*( ) - moveDot*( ) from
     TextAreaBasicModel_extended:Cursor, setColumn( int newColumn ),
     offsetColumn( int offset ), setLine( int line );
    Document.resetMarkAndProceed( ) => createLineBreak( ); /* should also
     be converted to insertString( String s ), removeCharacter( )
     and appendNextToCurrentLine( ) */
};

DocumentMonitoring := AbstractDocumentMonitoring + Monitoring{
    Speaker => Document;
    Document.listeners => Document.docListeners;
    Listener => simplenotepad.event.DocumentListener;
    Document.fire_eventHappened( ) => fire_documentChanged( );
    simplenotepad.event.DocumentListener.eventHappened( ) => documentChanged( );
    Document.addListener( simplenotepad.event.DocumentListener ) =>
     addDocumentListener( simplenotepad.event.DocumentListener );
};

TextAreaObservableModel := TextAreaModel + DocumentMonitoring{
    Document.change( ) => moveCursor*( ) from TextAreaModel:Document,
     offsetCursorColumn( int ), setCursorColumn( int newColumn ),
     insertString( String s ), removeCharacter( ), createLineBreak( ),
     appendNextToCurrentLine( ),
     clear( ),
     setCursorDotColumn( int newDotColumn ), removeSelectedText( ),
     cut( ), paste( );
};
```

Terão que ocorrer adições de temas em dois níveis diferentes. Na fórmula de composição que gera o tema **TextAreaObservableModel**, terá de ser feita a adição de um tema **ClipboardMonitoring** cujo objetivo é conter a implementação do registro para monitorar as mudanças de estado na área de transferência.

```
ClipboardMonitoring := AbstractClipboardMonitoring + Monitoring{
    Speaker => Document;
    Document.listeners => Document.clipboardListeners;
    Listener => simplenotepad.event.ClipboardListener;
    Document.fire_eventHappened( ) => fire_clipboardChanged( );
    simplenotepad.event.ClipboardListener.eventHappened( ) =>
     clipboardChanged( );
    Document.addListener( simplenotepad.event.ClipboardListener ) =>
     addClipboardListener( simplenotepad.event.ClipboardListener );
};

TextAreaObservableModel := TextAreaModel + DocumentMonitoring{
    Document.change( ) => moveCursor*( ) from TextAreaModel:Document,
     offsetCursorColumn( int ), setCursorColumn( int newColumn ),
     insertString( String s ), removeCharacter( ), createLineBreak( ),
     appendNextToCurrentLine( ),
     clear( ),
     setCursorDotColumn( int newDotColumn ), removeSelectedText( ),
     cut( ), paste( );
} + ClipboardMonitoring{
    Document.changeClipboard => cut( ), copy( );
};
```

Em outro nível, o tema **TextAreaModel** terá de ser complementado por um tema **TextAreaClipboardModel** que implementa as novas rotinas da classe *Document* que não são responsáveis por monitoramento de mudanças de estado, e a classe *Clipboard* usada pela *Document* para implementar a área de transferência.

```
TextAreaModel := TextAreaBasicModel_extended + TextAreaSelectionModel{
    Cursor.proceedAndResetMark( ) => move*( ) - moveDot*( ) from
     TextAreaBasicModel_extended:Cursor, setColumn( int newColumn ),
     offsetColumn( int offset ), setLine( int line );
    Document.resetMarkAndProceed( ) => createLineBreak( ); /* should also be
     converted to insertString( String s ), removeCharacter( )
     and appendNextToCurrentLine( ) */
} + TextAreaClipboardModel;
```

O diagrama de classes correspondente ao tema **TextAreaClipboardModel** é apresentado na Figura 4.16.

A seção anterior e esta seção demonstram uma necessidade importante quando se evolui um programa no DSOT. Antes de implementar qualquer nova característica do programa, o desenvolvedor deve primeiramente determinar o escopo (nível) da alteração: em uma análise descendente (*top-down*) que se inicia do tema de mais alto nível que corresponde ao programa, qual é o tema composto de menor nível na "hierarquia" de temas que deve ser alterado por meio da adição de um tema que o complementa? Isso requer um bom conhecimento do *design* anterior do programa. Um erro na escolha do nível pode afetar negativamente o *design* do programa, portanto bastante cautela deve ser tomada. Isso demonstra que o desenvolvedor deve conhecer bem o propósito de cada tema antes de efetuar uma alteração em como os temas são compostos para originar o programa. Talvez seja necessário haver uma forma de refatoração para tornar o *design*

mais adequado caso tal o desenvolvedor cometa acidentalmente um erro na escolha do nível, o que é deixado como um trabalho futuro.

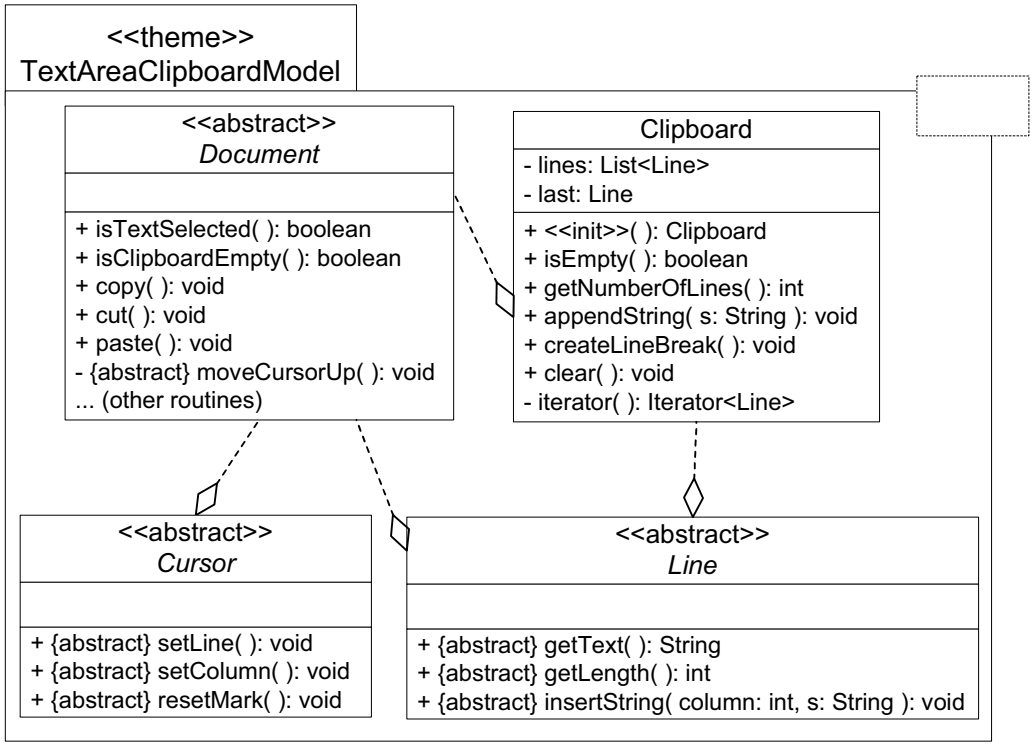


Figura 4.16: Diagrama de classes do tema **TextAreaClipboardModel**.

4.5.5 Mudança invasiva no tema TextAreaBasicModel para se adequar ao item de menu "New"

Conforme discutido no final da Seção 4.5.3, o tema **TextAreaClearIndirectControler** contém a implementação da rotina *clear( )* da classe *SimpleTextArea*. Esta rotina simplesmente delega a invocação de *clear( )* à classe *Document*, que não é implementada naquele tema. Essa rotina poderia ser fornecida em um novo tema do modelo, digamos **TextAreaClearModel**. Contudo, a criação de um tema tão pequeno e a alteração das fórmulas de composição para encaixar o novo tema acarretariam mais complexidade do que simplesmente adicionar invasivamente esta rotina ao tema **TextAreaBasicModel**. Mais um motivo para adicioná-la ao tema **TextAreaBasicModel** seria que a rotina não utiliza nenhuma informação além daquela manipulada pelo tema referido. A rotina *clear( )* não utiliza, por exemplo, informação a respeito da eventual seleção de fragmentos do documento nem a respeito da área de transferência. A inconsciência dos módulos é um valor desejável, mas não é absoluto; compromissos no *design* podem considerar que ocasionalmente a inconsciência seja violada para se atingirem outros benefícios.

Uma imagem de tela do programa obtido nesta iteração aparece na Figura 4.13.

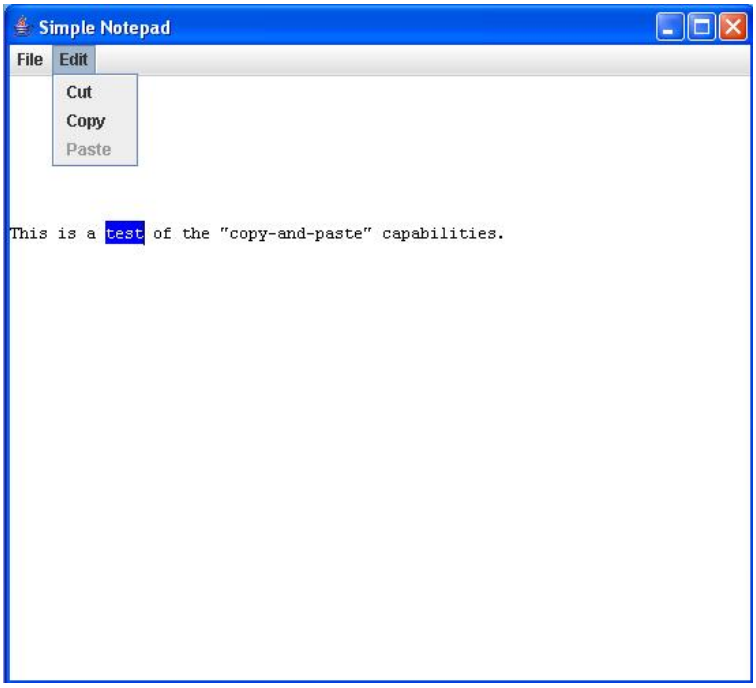


Figura 4.17: Imagem de tela do protótipo executável obtido na terceira iteração.

4.6 Considerações Finais

Este capítulo demonstrou que é possível para um indivíduo desenvolver um sistema de software utilizando DSOT em um processo baseado em iterações. O próximo capítulo analisará se é viável utilizar DSOT para o desenvolvimento de software na prática. Para isso, DSOT terá de ser comparado com outras abordagens existentes para averiguar se seus benefícios são maiores que suas desvantagens.

CAPÍTULO

5

Avaliação

5.1 Considerações Iniciais

O capítulo 4 demonstrou por meio de um estudo de caso como as idéias do DSOT podem ser postas em prática no contexto de um projeto real. Neste capítulo discute-se a respeito das vantagens e desvantagens que os mecanismos de DSOT oferecem para o desenvolvimento de software, em comparação com outras abordagens existentes.

5.2 Comparação entre DSOT e POO

Em POO, a herança e o envolvimento (*wrapping*) são os mecanismos utilizados para estender um módulo de forma não-invasiva. Contudo, os dois apresentam a desvantagem de que, quando usados, requerem mudanças invasivas nos módulos clientes. Além disso, o uso abusivo de herança fere o propósito fundamental de tal mecanismo, que é criar subtipos e não efetuar extensões *ad hoc* em uma classe.

Em DSOT aparece o mecanismo de composição em temas, que permite estender um módulo de forma não-invasiva e não requer mudanças invasivas nos módulos clientes. Isso parece sugerir que a abordagem de DSOT potencialmente reduz a frequência do uso de herança e de envolvimento, já que, para alguns cenários de uso de herança e de envolvimento típicos em POO, a composição de temas ofereceria uma melhor solução. Deve ser investigado futuramente em quais cenários a composição é mais adequada do que a herança e o envolvimento.

Outra vantagem do DSOT em relação ao POO é o maior suporte ao reúso. Em POO, basicamente se reúsam classes inteiras. Uma das poucas maneiras de reutilizar parte de uma classe é refatorá-la em duas, em que uma é superclasse da outra, para que se possa herdar da implementação parcial que se deseja reusar. Todavia, além dessa solução exigir uma mudança invasiva, nem sempre isso é possível ou adequado. Em DSOT, além de classes, temas inteiros também podem ser reutilizados. Um tema pode também conter parte da implementação de uma classe, assim, ao reutilizar o tema, o desenvolvedor de fato reutiliza aquela implementação parcial.

A principal vantagem do POO sobre o DSOT é que em POO não há a possibilidade de interação indesejada entre temas, já que não existem temas. Em DSOT, é realmente complicado verificar se ocorrem interações indesejadas entre temas, principalmente em sistemas maiores nos quais existem muitos temas que participam de composição. Um fator que reduz um pouco mas não elimina este problema é a possibilidade de visualizar o resultado da composição como se fosse um tema primitivo. Mas é muito necessário que se estabeleçam mais diretrizes de *design* para minimizar a possibilidade de efeitos colaterais indesejados na interação entre temas, já que em DSOT a composição é o mecanismo fundamental de formação de um programa.

Especula-se que a abordagem de DSOT tenha as seguintes vantagens potenciais sobre POO:

- *Simplicidade*: Aparentemente é mais fácil desenvolver vários temas separados do que desenvolver uma estrutura de classes universal para o programa completo. As estruturas de classes de cada tema são individualmente mais simples do que uma estrutura de classes universal, já que cada uma delas delas lida com somente um interesse estrutural, em vez de lidar com todos os interesses relevantes de uma só vez. Enquanto no POO cada tipo de dados (classe) é definido por todas as operações que podem agir sobre suas instâncias para o programa completo, no DSOT a definição do tipo de dados contém somente as operações necessárias para realizar o propósito de um tema. Isto é, o DSOT reconhece explicitamente que a definição de um tipo é relativa ou subjetiva a cada tema que auxilia a compor o programa (Harrison e Ossher, 1993).

- *Compreensão*: Os relacionamentos de composição fornecem visibilidade explícita à arquitetura do programa e ao processo de conexão entre os módulos do sistema. A quantidade e complexidade dos relacionamentos de composição também sugerem o nível de complexidade geral do programa. Também é mais fácil compreender a implementação correspondente a um interesse estrutural localizada em um único tema do que compreender aquela implementação espalhada ao longo de todo o programa, uma consequência inevitável do POO.

- *Evolução*: Com o DSOT, as mudanças possuem um impacto relativamente mais limitado na estrutura geral do software, já que alguns tipos de mudança são relacionados a somente um interesse estrutural, ou a um pequeno conjunto de interesses estruturais. A modificação de várias classes aparentemente não-relacionadas pode, com um *design* adequado e alguma sorte, ser substituída pela adição de um novo tema. Além disso, é oferecida a possibilidade

de compor componentes distintos de maneiras mais flexíveis do que o POO permite. O DSOT também está um passo mais próximo da programação diferencial e da adaptabilidade não-invasiva (Czarnecki e Eisenecker, 2000). No POO, as opções para praticar programação diferencial se restringem à herança e ao envolvimento (*wrapping*), e ambos requerem mudanças invasivas nos clientes dos módulos. Com o DSOT, podem-se efetuar composições que não requerem mudanças invasivas nos clientes.

- *Reúso*: O potencial de reúso é melhorado porque com o DSOT é possível desenvolver classes mais simples, já que a implementação de uma classe em um tema contém parte da implementação de um único interesse estrutural. Além do mais, o aparecimento de um novo tipo de módulo, o tema, possibilita a criação de temas reusáveis, um conceito que não existia no POO. Uma outra possibilidade é o reuso de relacionamentos de composição: alguns deles podem ser padrões, aplicáveis a vários tipos de software.

Outra diferença entre DSOT e o POO é a implementação interna de classes. No DSOT, há o conceito de rotinas incompletas (que possuem uma instrução *proceed*) e é possível a instanciação de classes abstratas. Tais mecanismos são considerados inválidos no POO, conforme explicação feita na Seção 3.2.1.

5.3 Comparação entre DSOT e DSOA com *AspectJ*

A abordagem de DSOA com *AspectJ* (The AspectJ Team, 2006) define um novo tipo de módulo, o aspecto, para modularizar a implementação de interesses estruturais entrecortantes. No DSOA com *AspectJ*, o software é majoritariamente estruturado por meio de classes, e a implementação relativa a interesses estruturais secundários é removida da estrutura-base (a estrutura de classes) e localizada em aspectos. Os aspectos são então compostos com a estrutura-base para formar o programa completo.

As idéias que culminaram no *AspectJ*, como os conceitos de interesses diferentes agindo sobre um mesmo módulo, espalhamento e entrelaçamento, causaram um grandioso avanço na compreensão dos problemas do POO. Esse novo grau de compreensão foi fortemente aproveitado como base do DSOT. A motivação da abordagem de DSOA *AspectJ* é fundamentalmente a mesma motivação para o DSOT. Entretanto, DSOT evolui a compreensão em alguns pontos que não foram percebidos pelos criadores do *AspectJ*.

A percepção que faltou aos criadores da abordagem de DSOA com *AspectJ*, na época, foi que *as partes de implementação referentes a dois ou mais interesses diferentes em uma mesma rotina de uma classe continuam invariavelmente sendo parte da rotina, mesmo se o desenvolvedor desejar visualizar a estrutura do software em termos de interesses estruturais*.

O que isso significa é que, quando um desenvolvedor modulariza as partes de implementação de um interesse estrutural em um aspecto, ele está ignorando que aquelas partes de implementação

devem continuar pertencendo à rotina da qual aquelas partes de implementação foram extraídas. Em *AspectJ*, em vez de pertencerem à rotina original, aquelas partes de implementação pertencem a um ou mais adendos do aspecto. Obviamente, o conjunto de junção associado àqueles adendos indica a rotina que será "afetada", que é a rotina de onde as partes de implementação foram extraídas, mas isto não significa o mesmo que afirmar que aquela implementação pertence à rotina original. Para ilustrar este argumento, compare a implementação do *logging* em *AspectJ*, a seguir, com a implementação do *logging* no DSOT, mostrada na Seção 3.2.

```
public aspect Logging{
  private pointcut loggedRoutines( ): execution( void A.foo( ) ) ||
   execution( A.bar( ) ) || execution ( B.rtn( ) );

  around( ): loggedRoutines( ){
    logEntry( );
    proceed( );
    logExit( );
  }

  private void logEntry( ){
    ... //implementation
  }

  private void logExit( ){
      ... // implementation
    }
}
```

Em *AspectJ*, a implementação relativa ao interesse estrutural de *logging* pertence inteiramente ao aspecto **Logging**, mas não pertence às classes *A* e *B* (embora tais classes sejam afetadas pelo aspecto). Em DSOT, a implementação relativa ao interesse estrutural de *logging* pertence inteiramente ao tema **Logging**, mas também pertence às classes *A* e *B*; basta alternar a visualização entre a dimensão de interesses estruturais e a dimensão de tipos de dados para observar esse fenômeno.

Esta falta de percepção tem acarretado várias dificuldades para os desenvolvedores em observar e identificar efeitos colaterais indesejados na interação entre os módulos do software quando se utiliza *AspectJ*, dificuldades estas que se reduzem sensivelmente quando se utiliza DSOT. De qualquer forma, seria interessante em trabalhos futuros investigar se as diretrizes de *design* e os *idioms* criados para reduzir os problemas de interação entre os módulos de um programa desenvolvido em *AspectJ* ampliam ainda mais as capacidades de visualização do DSOT.

O modelo do *AspectJ* é baseado no conceito de interceptação em tempo de execução do fluxo de controle *default* do programa. A idéia de interceptação não é ruim, mas muito do que é feito por interceptação dinâmica pode ser feito por mecanismos estáticos em DSOT. Isso é importante porque os efeitos colaterais de um mecanismo que funciona em tempo de execução (e portanto afeta o comportamento mas não a estrutura diretamente visível) são mais complexos de visualizar do que os efeitos colaterais de uma interação estática entre elementos do programa (que afeta a estrutura diretamente visível do programa e, por consequência, o comportamento). Assim, quando possível, é melhor utilizar um mecanismo estático do que um mecanismo dinâmico, pois o es-

tático possui como vantagens: maior compreensibilidade, já que é mais fácil visualizar o efeito de um mecanismo que que afeta diretamente a estrutura do programa, e não somente o comportamento; melhor desempenho, pois muitas otimizações podem ser feitas por um compilador por meio da análise da estrutura do programa; e possibilidade de verificação automática e detecção mais precoce de defeitos por um analisador estático, defeitos tais que de outra forma apareceriam em alguns cenários em tempo de execução, dificultando a depuração e aumentando a possibilidade de um defeito ser encontrado somente quando o usuário manipular o programa de alguma forma que a equipe de testes não considerou.

O modelo de execução do *AspectJ* sugere que ocorre uma interceptação quando a rotina afetada é invocada ou executada, dependendo da tempo escolhido pelo desenvolvedor do aspecto para a interceptação. Ocorre uma dissonância entre o modelo de execução e a maneira como a interceptação é realmente implementada pelo compilador do *AspectJ*. Quando a interceptação deve ser feita antes, depois ou ao invés da execução de uma rotina, o compilador insere o código de interceptação do adendo nos *bytecodes* (código resultante da compilação) da rotina afetada[1]. Este modelo de execução funciona, mas é mais complexo do que o necessário porque:

- Considera alteração dinâmica de comportamento em vários contextos que poderiam simplesmente considerar alteração estática de estrutura e, por consequência, de comportamento.
- Difere da maneira como a implementação pelo combinador é realmente feita. No DSOT, em contraste, não há um novo modelo de execução; o programa gerado ao final de todas as composições necessárias é orientado a objetos. Além disso, o modelo de composição não é um modelo de execução porque as composições ocorrem estaticamente. A ferramenta de composição atuaria da maneira que o desenvolvedor realmente espera. O usuário da abordagem (desenvolvedor) poderia inclusive visualizar o resultado da composição para se assegurar de que a composição foi efetuada da maneira desejada.
- Se a estratégia de implementação real pelo combinador se concentrar apenas em obter maior otimização do código compilado e ignorar a necessidade eventual de depuração de programas, a distinção entre o modelo conceitual de execução e o modelo real de implementação pelo combinador poderá causar inconveniências para o desenvolvedor durante o processo de depuração de programas.

Vários mecanismos do *AspectJ*, como os conjuntos de junção que envolvem *cflow*, podem fazer com que o *AspectJ* seja encarado também como uma forma mais disciplinada de utilizar reflexão. Diferentemente do que ocorre para o *AspectJ*, não são definidos mecanismos de reflexão dinâmica para o DSOT. Assim, em DSOT puro, os mecanismos de reflexão são simplesmente delegados a uma interface de programação de aplicações (*application programming interface*, API) composta

[1]Há também outras estratégias que o compilador utiliza, como por exemplo implantar um novo carregador de classes (*class loader*) no programa. Mas de qualquer forma o modelo de execução adotado pelo desenvolvedor continua sendo diferente da estrutura subjacente do programa após a compilação.

por classes e rotinas que fornecem alguma maneira de acessar informações sobre o programa em tempo de execução, como na solução tradicional no POO. Neste sentido o *AspectJ* possui uma vantagem importante sobre o DSOT. Mas não há nenhuma limitação teórica conhecida que impeça o uso simultâneo de *AspectJ* e de uma linguagem de programação que implemente DSOT.

Em *AspectJ*, é possível "desativar" rotinas utilizando um adendo do tipo *around( )* sem executar uma chamada *proceed( )* interna ao adendo. Não foi definido neste trabalho um operador de sobreposição de rotinas, classes e atributos obsoletos para DSOT por motivos explicados na Seção 5.6.

O DSOT contempla somente um tipo de módulo de primeira ordem, o tema, enquanto o *AspectJ* considera dois tipos de módulo de primeira ordem, o aspecto e a classe. O fato de possuir somente um tipo de módulo de primeira ordem confere ao DSOT maior homogeneidade e simplicidade.

No POO e no DSOT, toda rotina pertence a um tipo de dado[2], pois um conjunto de rotinas relacionadas é um dos principais elementos na definição de um tipo de dado. A abordagem de AspectJ destrói esta premissa porque os adendos (rotinas anônimas de aspectos) não pertencem a nenhum tipo de dado em particular, já que aspectos não são classes. Essa característica de *AspectJ* também fere a homogeneidade e simplicidade da abordagem.

Se o DSOT possui tantas vantagens sobre o DSOA com *AspectJ*, por que o segundo é mais bem sucedido do que o primeiro? Há explicações relacionadas à teoria e à prática para isso. A primeira razão teórica é que o DSOA *AspectJ* permite modularizar muitos outros tipos de interesse que o DSOT não consegue modularizar, em particular aqueles que envolvem alguma forma de informação obtida por meio de reflexão. Por exemplo, um comportamento encapsulado por um adendo *AspectJ* que afeta pontos de junção descritos por um conjunto de junção do tipo *call* ou *cflow* raramente pode ser encapsulado de forma elegante em DSOT. Em segundo lugar, o campo de DSOA com *AspectJ* está melhor estabelecido e mais maduro, por ter sido pioneiro e ter atraído atenção por mais tempo; o DSOT é mais recente e muito menos maduro para uso prático. Finalmente, o DSOA com *AspectJ* requer uma mudança de paradigma menos radical com relação ao POO do que o DSOT; por exemplo, classes são mantidas como módulos de primeira ordem no DSOA com *AspectJ*.

Mas a observação da infra-estrutura prática para o DSOA com *AspectJ* é que torna mais claro porque o DSOA com *AspectJ* tem assegurado maior sucesso. A abordagem de DSOA com *AspectJ* tem tido ferramentas mais maduras para apoiar a Engenharia de Software prática, acompanhadas de documentação de alta qualidade para a aprendizagem, como evidenciado pelo uso relativamente disseminado do combinador do *AspectJ* e de ferramentas relacionadas. O DSOT alcançará um estágio maduro comparável ao DSOA com *AspectJ* somente quando ferramentas maduras de DSOT surgirem para apoiar a Engenharia de Software prática.

As abordagems de DSOT e de DSOA com *AspectJ* são consideradas distintas, mas com um bom potencial de complementaridade mútua. Primeiramente, alguns resultados de pesquisa de uma

[2]Em POO puro isso acontece. Ignoram-se aqui idiossincrasias de algumas linguagens de programação específicas (por exemplo, *C++*) que ferem o modelo do POO, como a existência de rotinas externas a classes.

abordagem podem alimentar a pesquisa na outra abordagem, já que ambas compartilham vários conceitos em comum, como interesses estruturais, e motivações, como fornecer uma solução que supera as limitações do POO. Além disso, pode haver casos específicos nos quais uma abordagem é mais adequada do que a outra. Isso encorajaria o uso conjunto das duas abordagens, possivelmente originando uma nova abordagem que combina o melhor de ambas. É interessante especular sobre as várias combinações possíveis entre DSOT e DSOA com *AspectJ*, tais como: um tema que contenha tanto classes como aspectos; um aspecto não contido em um tema que afeta alguns temas; e assim por diante.

5.4 Comparação entre DSOT e *MDSoC/Hyperspaces*

Conforme comentado na Seção 3.2.2, o DSOT é basicamente uma evolução da abordagem de *MDSoC/Hyperspaces*. Há mais semelhanças do que diferenças entre as duas abordagens. Uma semelhança importante é a descrição de fórmulas de composição externamente aos módulos. Algumas falhas da abordagem original de *MDSoC/Hyperspaces* são reparadas, e alguns novos conceitos são incorporados. O DSOT possui uma terminologia diferente do *MDSoC/Hyperspaces*, mas os conceitos realmente importantes do *MDSOC/Hyperspaces* sobrevivem no DSOT com outros nomes. Em vários casos, os nomes dos conceitos foram alterados porque uma idéia por trás dos nomes antigos não tem mais sentido. Por exemplo, “hiperfatia” e “tema” são o mesmo conceito com nomes diferentes. Neste caso, o nome foi alterado porque “hiperfatia” se refere a algo presente no hiperespaço, e hiperespaço é um conceito dispensável para o DSOT. O desenvolvedor precisa saber que existem duas dimensões relevantes, de tipos de dados e de interesses estruturais, portanto não há motivação prática para se definir “hiperespaço” (espaço de várias dimensões) em DSOT.

Uma falha do *MDSoC* é tratar dois conceitos importantes, porém distintos, como se fossem um só: categorias de interesses estruturais e dimensões de visualização da implementação do programa. As categorias de interesses estruturais (características, propriedades, unidades de mudança, variabilidades, papéis, configurações, políticas, mecanismos, etc.) ajudam a organizar a implementação que aparece na dimensão de visualização de interesses estruturais, assim como pacotes em *Java* ajudam a organizar o conjunto de classes do programa. Mas não há vantagem clara ou necessidade em fazer com que as implementações correspondentes a interesses estruturais diferentes apareçam em dimensões de visualização distintas de interesses estruturais. A implementação relativa a um interesse estrutural individual pode ser analisada separadamente, da mesma forma que uma classe pode ser visualizada individualmente. Um tema de uma categoria pode ser composto com outro tema de outra categoria, desde que isso faça sentido no contexto do *design* específico daquele programa. Somente duas dimensões de visualização são necessárias, a de interesses estruturais e a de tipos de dados. Portanto, o DSOT poderia ser visto como uma espécie de abordagem de Separação Bidimensional de Módulos (por temas em uma dimensão e por classes em outra dimensão).

A comparação mais interessante entre as duas abordagens envolve a linguagem descrita neste trabalho para o DSOT e as linguagens de programação reconhecidas pelas ferramentas *Hyper/J* e *CME*. A semântica da composição é diferente entre DSOT e as demais abordagens.

Hyper/J possui o operador de composição *mergeByName* para fazer com que as unidades que possuem o mesmo nome nas duas hiperfatias envolvidas na composição sejam correspondentes. O *CME* possui o operador *merge concerns* que faz o mesmo. No DSOT, o operador + tem essa função.

Em *Hyper/J* há também o operador *nonCorrespondingMerge*, que faz com que unidades com o mesmo nome em hiperfatias diferentes não sejam correspondentes. No CME isso pode ser simulado com o operador *extend concern with*. Em DSOT, é necessário efetuar uma redefinição com o operador *{}* para alterar o nome de unidades que não devam corresponder, e então se aplica o operador + para efetuar a composição entre os temas.

O operador *overrideByName* de *Hyper/J* serve para sobrepor unidades que foram definidas anteriormente mas não são mais necessárias ou foram substituídas por implementações mais convenientes durante a evolução do software. No CME, o operador *override method with*, que age juntamente com o *merge concerns*, serve para o mesmo propósito. Não foi definido neste trabalho um operador de sobreposição de rotinas, classes e atributos obsoletos para DSOT por motivos dados na Seção 5.6.

Em DSOT, é possível redefinir unidades de um tema e em seguida efetuar a composição com outro tema com o uso do operador *{}* e em seguida o uso do operador +. Em *Hyper/J*, isso pode ser feito com um operador *equate* dentro de uma composição efetuada com *mergeByName*, ou *equate* dentro de uma composição efetuada com *nonCorrespondingMerge*. Em CME, se uma redefinição seguida de uma composição for desejada, não se pode utilizar o operador *merge concerns*; deve ser utilizado o operador *extend concern with* para permitir que unidades de um módulo de interesse sejam redefinidas e em seguida uma composição ser efetuada. Isso significa que o CME não adota um *design* ortogonal de mecanismos para sua linguagem de composição, o que é desvantajoso, pois não há um operador de redefinição independente de um operador de composição. Portanto, DSOT repara esta falha de *design* do CME.

Nem *Hyper/J* nem CME possuem mecanismos para permitir a uma rotina efetuar a complementação de outra rotina por meio de uma anexação simultânea de implementação antes e depois da rotina original. Isso é considerado uma falta importante, pois, como se pode observar no Capítulo 4, em vários momentos necessita-se de um mecanismo desse tipo. Em DSOT isso pode ser feito por meio de uma rotina incompleta, que contém uma instrução *proceed*.

Em suma, os operadores + e *{}* juntamente com a instrução *proceed* permitem fazer tudo o que os operadores de *Hyper/J* e *CME* fazem, à exceção da sobreposição de elementos de um tema, que necessitaria de um operador novo. Mas a instrução e os dois operadores mencionados permitem também fazer coisas que não são possíveis em *Hyper/J* e *CME*, como a complementação de rotinas antes e depois simultaneamente. Além disso, o *design* dos operadores do DSOT são ortogonais, isto é, podem ser usados simultaneamente ou separadamente, agregando poder à abordagem.

5.5 O *Design* do Programa *SimpleNotepad* é bom o suficiente?

A resposta da pergunta que aparece no título desta seção depende do que significa um *design* bom o suficiente. A qualidade de um *design* é difícil de ser medida porque envolve, dentre outras coisas, fatores subjetivos como questões estéticas e artísticas, embora seja possível discutir alguns valores que permitem sugerir quão bom é um *design*.

Uma dimensão importante para a qualidade de um *design* é a evoluibilidade. Poderia-se afirmar que um *design* possui boa evoluibilidade se ele não sofrer muitas mudanças invasivas após um processo de evolução no qual determinados requisitos são considerados. Essa maneira de medir evoluibilidade possui dois problemas: ela exige que se modifique o *design*, o que nem sempre é viável, e ela também fornece um grau de evoluibilidade relativo aos requisitos escolhidos para a evolução, e não absoluto. Mesmo com tais problemas, essa definição é suficiente para os propósitos deste trabalho.

O *design* das duas primeiras iterações parece ter uma evoluibilidade razoável, porque poucos módulos tiveram de ser invasivamente alterados na iteração seguinte, e as alterações não foram relativamente complicadas. O elemento de software mais sujeito a alterações foi o conjunto de fórmulas de composição. Várias fórmulas de composição tiveram de ser refatoradas ou complementadas para atualizar a maneira de gerar alguns temas compostos. Com relação aos módulos, a evoluibilidade é razoável graças à estruturação do *design* em torno de padrões arquiteturais relativamente estáveis.

De acordo com a maneira de medir evoluibilidade indicada nesta seção, não é possível afirmar se o *design* obtido na terceira iteração possui boa evoluibilidade, já que ele não foi sujeito a modificações. Todavia, como os mesmos princípios para criar o *design* das duas primeiras iterações foram seguidos, especula-se que sua evoluibilidade também seja razoável.

Não há *design* perfeito, pois todo *design* é obtido por meio de compromissos que exigem sacrificar um pouco determinada característica de qualidade desejada para se obter outra. Mas sempre é possível melhorar o *design* existente, pois em um projeto real, devido a restrições de tempo, algumas melhorias não são feitas porque o tempo é alocado para tarefas mais prioritárias. As melhorias descritas a seguir, por exemplo, poderiam ser feitas sobre o *design* do *SimpleNotepad*, já que não foram feitas ao longo das três iterações.

O construtor da classe *SimpleTextArea* da maioria dos temas que compõem **TextAreaController** segue o seguinte padrão:

```
abstract public class SimpleTextArea{
   public SimpleTextArea( Document doc ){
      Action xxxAction = new XXXAction( doc );
      this.getInputMap( ).put( KeyStroke.getKeyStroke( "some keystroke" ),
       xxxAction.getValue( Action.NAME ) );
      this.getActionMap( ).put( xxxAction.getValue( Action.NAME ),
       xxxAction );
   }

   abstract private InputMap getInputMap( );

   abstract private ActionMap getActionMap( );
   }
```

Ocorre o fenômeno de replicação de código em duas linhas, que requerem o uso de *xxxAction.getValue(Action.NAME)*. Além disso, a declaração das duas rotinas abstratas privadas estão replicadas (espalhadas) em todos os temas que seguem esse padrão. Para eliminar essa replicação, basta refatorar o *design* de forma a obter o seguinte.

```
//refactored theme
abstract public class SimpleTextArea{
   public SimpleTextArea( Document doc ){
      this.mapKeyStrokeToAction( "some keystroke",
       new XXXAction( doc ) );
   }

   abstract private void mapKeyStrokeToAction( String description,
    Action action );
}
```

```
//theme KeyStrokeToActionMapping (new theme)
abstract public class SimpleTextArea{
   private void associateKeyStrokeWithAction( String description,
    Action action ){
      String actionName = action.getValue( Action.NAME );
      this.getInputMap( ).put( KeyStroke.getKeyStroke( description ),
       actionName );
      this.getActionMap( ).put( actionName, action );
   }

   abstract private InputMap getInputMap( );

   abstract private ActionMap getActionMap( );
}
```

Um novo tema, denominado **KeyStrokeToActionMapping**, passa a conter a implementação necessária aos vários temas que seguiam o padrão anterior, e portanto aparecerá também como argumento na fórmula de composição que gera **TextAreaController**. Dessa forma, se elimina uma forma de replicação de código, tornando o *design* mais simples e reduzindo a quantidade total de linhas de código do programa.

Outro tipo de alteração que pode ser feito para melhorar o *design* do programa é reduzir o número de linhas de implementação de algumas rotinas, como por exemplo *copy( )* e *paste( )* da

classe *Document* do tema **TextAreaClipboardModel**. São necessárias para isso algumas refatorações que remanejam responsabilidades para outras classes. Felizmente, poucas rotinas possuem esse problema. Em geral, as rotinas do programa possuem cerca de cinco linhas de instruções, incluindo o código da pré-condição.

Note que as melhorias sugeridas para o *design* obtido na terceira iteração são de natureza local, isto é, as refatorações necessárias não acarretariam uma mudança drástica no *design* do sistema. Não foi detectada até o momento de redação desta dissertação nenhuma possível melhoria de alcance mais global. Isso evidencia que ou 1) o *design* atual está suficientemente bom, embora não esteja ótimo, ou 2) o desenvolvedor não está adequadamente capacitado para encontrar problemas no *design*. A primeira hipótese parece mais provável, entretanto isso teria de ser confirmado por meio de uma auditoria profissional feita por terceiros.

As técnicas de refatoração podem ser encaradas como maneiras disciplinadas de efetuar mudanças infelizmente invasivas. O *design* final do *SimpleNotepad*, após todas as composições serem feitas, é muito aproximado ao *design* que seria obtido no POO. Contudo, o desenvolvimento foi muito mais facilitado porque várias refatorações que seriam feitas no POO não foram necessárias. Por exemplo, em vários casos, em vez de se alterar a implementação original de uma classe ou rotina, criava-se o complemento da classe em um novo tema. Isso não significa que DSOT elimina as necessidades de refatorações. Outras refatorações específicas ao DSOT são necessárias em alguns casos, como mover parte da implementação de uma rotina de um tema a outro. Além disso, refatorações para melhorar o *design* ou corrigir erros e omissões continuam sendo inerentes ao desenvolvimento de software. Mas alguns tipos de refatorações necessários em POO não são tão frequentemente efetuados em DSOT, o que já é um avanço não-desprezível.

5.6 Outras Observações Sobre a Abordagem de DSOT

Esta seção apresenta algumas informações importantes a respeito da abordagem de DSOT que não foram comentadas anteriormente neste trabalho.

Tarr et al. (1999) também notaram que as implementações correspondentes a interesses estruturais distintos raramente são ortogonais: elas interagem umas com as outras, e se sobrepõem em algumas classes e rotinas. É provável que haverá uma sobreposição significativa: na dimensão de tipos de dados, uma mesma classe pode conter parte da implementação correspondente a vários interesses estruturais. Da mesma forma, várias classes podem conter parte da implementação correspondente a um mesmo interesse estrutural. A sobreposição é aceitável; na verdade, a sobreposição é justamente a responsável por grande parte do poder da abordagem de DSOT.

Esta grande flexibilidade levanta a questão de como os desenvolvedores deveriam escolher temas para decompor um determinado programa, e se tal liberdade aumenta a probabilidade de erro e abuso. Para isso, são necessárias mais diretrizes de *design*. Mesmo com a existência de tais diretrizes, o uso de temas, como qualquer outro mecanismo de modularização, requer bom

julgamento para fazer compromissos adequados. Se decisões estruturais importantes se tornarem incorretas por causa de erros de *design* ou de mudanças dramáticas nos requisitos, a reestruturação do sistema pode se tornar necessária, como ocorre com a tecnologia convencional (Tarr et al., 1999).

Para fornecer suporte à composição de temas, é necessário haver uma ferramenta de composição capaz de aplicar as fórmulas de composição aos temas para gerar novos temas. A composição manual é conceitualmente possível, como este trabalho demonstra, mas é totalmente irrealista para o desenvolvimento profissional de programas (Tarr et al., 1999). Esse é o maior de todos os obstáculos que previnem o DSOT de ser utilizado na prática. O trabalho futuro mais prioritário é criar uma ferramenta de composição. A presença de tal ferramenta facilitaria até mesmo as pesquisas na área, pois evitaria a composição manual, que é demorada, entediante, propensa a erros e desencorajadora.

Não foi definido neste trabalho um mecanismo para substituir não-invasivamente uma parte obsoleta da implementação porque na implementação do estudo de caso tal mecanismo não foi necessário. É mais conveniente definir mecanismos novos em uma abordagem somente se surgir uma necessidade real, e não simplesmente teórica, para tais mecanismos, sob pena de poluir a abordagem com conceitos complexos e desnecessários. A existência de uma necessidade real assegura que não haverá uma definição equivocada na semântica do mecanismo. Mas não há nenhuma limitação teórica que impeça DSOT de possuir um operador desse tipo. Caso surja uma necessidade real em estudos de caso futuros, um operador de sobreposição será definido.

Dada sua ênfase em mecanismos estáticos, não está claro como DSOT poderia ser utilizado juntamente com linguagens de programação dinâmicas, como *Smalltalk* e *LISP*. Toda a idéia de DSOT foi concebida considerando-se cenários de desenvolvimento em linguagens estáticas como *Java* e *Eiffel*. Seria interessante analisar futuramente quais mecanismos definidos pelo DSOT são aplicáveis para linguagens dinâmicas, e como pode-se adaptar a abordagem para auxiliar o desenvolvimento de software em ambientes nos quais linguagens dinâmicas são utilizadas.

A questão da escalabilidade do DSOT ainda não está resolvida. Por um lado, a simplicidade da implementação de cada tema (referente a somente um interesse estrutural) pode tornar o desenvolvimento de software mais fácil. Por outro lado, a coordenação das fórmulas de composição é delicada, já que qualquer falha na definição das fórmulas de composição pode tornar difícil a detecção do problema. No DSOT, há dois lugares nos quais pode haver uma falha: dentro dos temas e nas fórmulas de composição. Assim, o desenvolvimento de sistemas de software complexos com equipes grandes em DSOT parece exigir algum tipo de coordenador geral de composição para evitar que desenvolvedores independentes causem não-intencionalmente problemas sérios ao alterarem as fórmulas de composição (Clarke e Baniassad, 2005).

A possibilidade de reúso de temas é uma vantagem importante do DSOT. Temas de vários níveis podem ser reusados, desde temas primitivos até temas que resultam da composição de outros temas compostos. No caso do sistema *Simple Notepad*, por exemplo, os temas **TextAreaBasicModel**, **TextAreaModel** e **BasicWindowSetup** são reusáveis em teoria. O tema **TextAreaBa-**

sicModel pode ser reutilizado para criar um sistema de edição de textos no qual a seleção de fragmentos do documento é dispensável. O tema **TextAreaModel** pode ser reutilizado para criar um sistema de edição de textos similar ao *Simple Notepad*, mas com uma diferente interface com o usuário. Já o tema **BasicWindowSetup** pode ser reutilizado em uma aplicação de manipulação de relatórios, já que ele só define uma janela e uma área de texto genérica. O reuso de temas também poderia facilitar o desenvolvimento de famílias de produtos de software porque algumas espécies de variabilidades e características comuns poderiam ser implementadas por meio de temas.

No estudo de caso descrito no Capítulo 4 foram encontrados dois cenários nos quais mudanças invasivas em temas primitivos foram necessárias, e vários cenários de mudanças invasivas em fórmulas de composição. É esperado que as fórmulas de composição sejam alteradas quando o programa evolui, porque normalmente são adicionados novos temas. Todavia, mudanças invasivas em temas já existentes tipicamente não são desejadas, mas podem ser necessárias.

O primeiro tipo de mudança invasiva é aquele que envolve remoção de parte da implementação de um tema para compartilhá-la com outro tema. Esse tipo de mudança foi observado na Seção 4.5.2. Naquele caso, desejava-se reusar a implementação da rotina *execute(Window)* da classe *ApplicationExitWindowListener* do tema **BasicWindowSetup**. A classe *ApplicationExitAction* do tema **ExitMenuItemInsertionInFileMenu** só foi capaz de se beneficiar daquela rotina porque ela foi removida de **BasicWindowSetup** e inserida em um novo tema, denominado **BasicApplicationExitPolicy**. Esse tema então poderia ser composto com os dois anteriores para fornecer a funcionalidade desejada.

O outro tipo de mudança invasiva é aquele que envolve inserção de novas rotinas em uma classe de um tema já existente a fim de não ser necessário criar um novo tema e assim tornar mais complicado o conjunto de fórmulas de composição. Isso corresponde a um compromisso entre simplicidade e inconsciência. Tal cenário foi observado quando se adicionou invasivamente uma nova rotina no tema **TextAreaBasicModel** para que houvesse suporte a uma funcionalidade que não existia na época em que **TextAreaBasicModel** foi criado (Seção 4.5.5).

Portanto, há cenários em que a abordagem de DSOT é utilizada e mesmo assim mudanças invasivas são inevitáveis. É importante investigar se é possível evitar tais mudanças invasivas com novos mecanismos, ou se realmente esses problemas são inerentes ao desenvolvimento de software em geral.

Na Tabela 5.1 é resumida a comparação entre a abordagem de DSOT e as outras abordagens comentadas neste capítulo.

5.7 Considerações Finais

Neste capítulo a abordagem de DSOT foi comparada com algumas outras abordagens de desenvolvimento de software, e vantagens, desvantagens e características especiais do DSOT foram discutidas. Pela gama de assuntos abordados, observa-se que o DSOT abre um vasto campo de

Característica / Abordagem	***POO***	***AspectJ***	***Hyper/J***	**CME**	**DSOT**
Módulo de primeira ordem	classe	aspecto e classe	hiperfatia	interesse	tema
Módulo de segunda ordem	rotina (aparece dentro de uma classe)	rotina (aparece dentro de uma classe ou aspecto) e rotina anônima ou adendo (aparece dentro de um aspecto)	classe (aparece dentro de uma hiperfatia)	classe (aparece dentro de um interesse)	classe (aparece dentro de um tema)
Módulo de terceira ordem	-	-	rotina (aparece dentro de uma classe)	rotina (aparece dentro de uma classe)	rotina (aparece dentro de uma classe)
Fórmula de composição descrita separadamente dos módulos	-	não	sim	sim	sim
Existência de mecanismos para complementar uma rotina em seu começo e fim simultaneamente	não	sim	não	sim	sim
Existência de mecanismos para substituir não-invasivamente uma parte obsoleta da implementação	não	sim	sim	sim	não (mas não há nada que impeça em princípio a criação de um mecanismo desse tipo)
Mecanismos de reuso	herança em classes, envolvimento (*wrapping*) em classes	herança em classes, envolvimento em classes, herança em aspectos (a partir de uma classe ou de um aspecto), composição em aspectos (com classes ou com outros aspectos)	herança em classes, envolvimento em classes, composição e redefinição em hiperfatias	herança em classes, envolvimento em classes, composição e redefinição em interesses	herança em classes, envolvimento em classes, composição e redefinição em temas

Tabela 5.1: Uma comparação entre as abordagens comentadas neste capítulo.

pesquisa. Assim, no próximo capítulo são descritas algumas conclusões obtidas durante esta pesquisa, e são feitas várias sugestões de trabalhos futuros.

CAPÍTULO

6

Conclusões e Trabalhos Futuros

6.1 Organização

Neste capítulo são comentadas as conclusões que foram obtidas ao longo do projeto de pesquisa que culminou com o surgimento desta dissertação. Também é sugerida uma lista de trabalhos futuros para que esta pesquisa possa ter continuidade.

O capítulo está dividido como se segue. Na Seção 6.2 são discutidas as contribuições científicas fornecidas por este trabalho. Finalmente, na Seção 6.3 é mostrada uma lista de trabalhos futuros sugeridos que teriam como base os resultados obtidos nesta pesquisa.

6.2 Contribuições

As contribuições mais relevantes apresentadas nesta dissertação são as seguintes:

- Especificação do modelo conceitual de uma abordagem, denominada Desenvolvimento de Software Orientado a Temas, que possui como objetivo superar algumas deficiências inerentes à estruturação de programas nas abordagens Paradigma de Orientação a Objetos, Desenvolvimento de Software Orientado a Aspectos com *AspectJ*, e Separação Multidimensional de Interesses em Hiperespaços com *Hyper/J* e CME. (Capítulo 3);

- Descrição de um estudo de caso que envolve a modelagem de *design* e a implementação de um sistema de software por meio do Desenvolvimento de Software Orientado a Temas em um processo iterativo (Capítulo 4);

Uma contribuição secundária desta pesquisa, a modificação da notação *Theme/UML* de modelagem de *design*, foi publicada em um evento da área. O texto de tal publicação encontra-se no Anexo A. No entanto, até o momento da redação desta dissertação, o autor ainda não havia publicado artigos que se concentram nas contribuições mais relevantes desta pesquisa.

É importante mencionar que o estudo de caso descrito neste trabalho e a comparação qualitativa feita entre as abordagens não demonstram de forma alguma que o DSOT é melhor do que o POO ou quaisquer outras abordagens, principalmente porque há muitos entraves para a utilização da tecnologia de DSOT na prática. No entanto, tais resultados oferecem motivação e evidências suficientes para que o DSOT seja melhor investigado por ser uma tecnologia bastante promissora.

6.3 Trabalhos Futuros

O trabalho de pesquisa que culminou nesta dissertação se manteve apenas na superfície do tópico. Há ainda muito o que ser explorado. Felizmente, o tópico aparenta ter um caminho promissor de futuros trabalhos de pesquisa. Acredita-se ser possível que a tecnologia de DSOT amadureça a ponto de se tornar prática para necessidades profissionais. Assim considerando, segue uma lista de sugestões de trabalhos futuros.

Em virtude da falta de uma ferramenta que efetua composição exatamente da maneira pretendida, foi necessário ao longo desta pesquisa efetuar as composições de temas manualmente, admitidamente uma tarefa entediante e propensa a erros. Assim, o principal obstáculo prático para o uso do DSOT é a falta de uma ferramenta de composição eficiente. Neste sentido, seria necessária a criação de um ambiente de desenvolvimento mínimo para DOST, que contém as seguintes ferramentas integradas: um editor para código-fonte de classes e fórmulas de composição, um compilador interativo e uma ferramenta de visualização e navegação que permita alternar entre a dimensão de tipos de dados e a dimensão de interesses estruturais. Como um passo inicial para se desenvolver tal ambiente, poderia ser criado um compilador não-interativo simples que permita compor temas. Seria interessante também, para fins de depuração e verificação, fazer com que a ferramenta de composição liste, quando solicitado, as classes, rotinas e variáveis de instância que estão sujeitas à combinação com elementos de outros temas.

É importante também investigar de forma mais aprofundada a interação entre contratos de rotinas e de classes quando temas são compostos. Uma sugestão para atingir este fim é examinar como o DSOT poderia se adequar a *Eiffel* como linguagem-base de programação, já que a abordagem descrita aqui foi pensada para *Java*, uma linguagem que não fornece suporte para o *design* por contratos (Meyer, 1997).

As questões de escalabilidade do DSOT merecem ser analisadas mais aprofundadamente. Em particular, parece ser necessário encontrar maneiras de modularizar os relacionamentos de composição, já que estes tendem a crescer à medida que o software se torna mais complexo. Também é importante efetuar estudos de casos com equipes de desenvolvedores trabalhando em conjunto

e utilizando DSOT, para descobrir que problemas são mais frequentes na criação de temas e na refatoração feita sobre fórmulas de composição para integrar temas novos.

Considerou-se neste trabalho um cenário que envolvia somente programação sequencial, sem estarem envolvidas *threads* e mecanismos de comunicação interprocessos. Assim sendo, como modularizar software por meio de temas tendo em vista a necessidade criar sistemas que envolvem concorrência e programação paralela?

A interação e evolução de temas em sistemas distribuídos também deve ser considerada. Por exemplo, é frequente o caso em que, quando um tema é inserido em um dos programas que compõem um sistema distribuído, temas devam ser adicionados também em outros programas?

A Engenharia de Linha de Produtos de Software é um ramo da Engenharia de Software que visa o desenvolvimento de famílias de produtos de software, isto é, conjuntos de produtos que compartilham certas características comuns (Weiss e Lai, 1999). As características que diferenciam os produtos de uma família são denominadas variabilidades. Um grande desafio é como representar e manipular variações entre os produtos de uma mesma família. Os campos de Engenharia de Linha de Produtos de Software e de DSOT convergem na necessidade de melhor separação de interesses (Murphy et al., 2001; Krueger, 2001). Seriam características comuns e variabilidades possíveis inspiradores de temas a serem criados? É importante também examinar a diferença entre a estrutura modular de um sistema no qual uma variabilidade é adicionada estaticamente e outro sistema no qual a mesma variabilidade é adicionada dinamicamente. Além disso, há de se verificar o impacto que o DSOT causa na análise de domínios e o desenvolvimento de linguagens de padrões.

A criação de *frameworks* orientados a temas seria algo bastante inovador. Existem *frameworks* orientados a aspectos, implementados em *AspectJ* (de Camargo e Masiero, 2005). O que aconteceria se eles fossem migrados para DSOT? Eles ficariam melhor estruturados? Muitas modificações à estrutura original seriam necessárias? Seriam mais fáceis de usar? Seria interessante comparar uma solução de *framework* orientada a temas com os *frameworks* orientados a aspectos que estão descritos na literatura.

Outro ramo no qual poderiam surgir pesquisas extensas seria a investigação de como deve ser feito teste quando o software é orientado a temas. Em particular, devem surgir diretrizes para se fazer testes de unidade, focando temas individuais e classes internas aos temas.

O campo de desenvolvimento de formalismos poderia encontrar um novo nicho de pesquisa com o DSOT. Um tipo abstrato de dados (TAD), conceito fundamental do POO, é bem definido formalmente: primariamente, ele se constitui de um identificador, um conjunto de rotinas relacionadas que são individualmente definidas conforme contratos que incluem pré-condições e pós-condições, e de outros elementos, como sua relação com outros tipos abstratos de dados. Contudo, interesse estrutural, um conceito fundamental do DSOT, não é tão bem definido até o momento. Sabe-se muito do que um interesse estrutural não é; por exemplo, critérios de qualidade, como confiabilidade e segurança, são interesses importantes de um sistema de software, mas não são

interesses estruturais conforme definido neste trabalho, já que eles não podem ser modularizados separadamente em um tema. Coleta de lixo e análise da tipagem de um programa também são interesses importantes, mas não são interesses estruturais no sentido que se aplica aqui. Sabe-se também algo sobre o que um interesse estrutural é: a implementação de um interesse estrutural corresponde à união de um subconjunto da implementação total de rotinas e atributos de várias classes. Mas falta ainda uma definição melhor de interesse estrutural, para que a sua teoria se torne tão madura quanto a teoria de TADs.

Como se percebe, a grande lista de trabalhos a serem feitos futuramente demonstra que ainda há muito a ser explorado para que a tecnologia de DSOT possa amadurecer de fato.

Referências Bibliográficas

BROOKS, F. P. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, v. 20, n. 4, p. 10–19, 1987.

DE CAMARGO, V. V.; MASIERO, P. C. Frameworks orientados a aspectos. In: *SBES'05: XIX Simpósio Brasileiro de Engenharia de Software*, Sociedade Brasileira de Computação, 2005.

CLARKE, S.; BANIASSAD, E. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, 2005.

CZARNECKI, K.; EISENECKER, U. W. *Generative programming: Methods, tools, and applications*. Addison-Wesley, 2000.

DIJKSTRA, E. W. *A discipline of programming*. Prentice-Hall, 1976.

ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-oriented programming: Introduction. *Commun. ACM*, v. 44, n. 10, p. 29–32, 2001.

FILMAN, R. E.; FRIEDMAN, D. P. Aspect-oriented programming is quantification and obliviousness. In: *ACM OOPSLA 2000 Workshop on Advanced Separation of Concerns*, 2000.

FINSETH, C. A. The craft of text editing: Emacs for the modern world. Online, disponível em `http://www.finseth.com/craft/` - último acesso em 14/03/2005, 1999.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.

HARRISON, W.; OSSHER, H. Subject-oriented programming: a critique of pure objects. In: *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, ACM Press, p. 411–428, 1993.

IBM RESEARCH Multi-Dimensional Separation of Concerns: Software Engineering using Hyperspaces. Online, available at `http://www.research.ibm.com/hyperspace/` - último acesso em 24/07/2005, 2000.

KAMINSKI, P. Applying multi-dimensional separation of concerns to software visualization. In: *ICSE 2001 Workshop on Advanced Separation of Concerns*, 2001.

KELLY, P. Text editor? What? Online, disponível em `http://web.greens.org/about/software/editor.html` - último acesso em 14/03/2005, 2005.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. Getting started with aspectj. *Communications of the ACM*, v. 44, n. 10, p. 59–65, 2001.

KICZALES, G.; LAMPING, J.; MENHDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J.-M.; IRWIN, J. Aspect-oriented programming. In: AKŞIT, M.; MATSUOKA, S., eds. *Lecture Notes in Computer Science - Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, v. 1241, Jyväskylä, Finnland: Springer-Verlag, p. 220–242, 1997.

KRUEGER, C. W. Using separation of concerns to simplify software product family engineering. In: *Proceedings of the Dagstuhl Seminar on Product Family Development, number 01161*, p. 71–77, 2001.

LADDAD, R. *AspectJ in Action: Practical Aspect-Oriented Programming*. 1 ed. Greenwich, Connecticut – USA: Manning Publications Company, 512 p., 2003.

LAI, A.; MURPHY, G. C.; WALKER, R. J. The tyranny of the file decomposition. In: *ICSE 2000 Workshop on Multi-dimensional Separation of Concerns*, 2000.

LISKOV, B.; GUTTAG, J. *Abstraction and specification in program development*. MIT Press, 1986.

MEYER, B. *Object oriented software construction*. 2 ed. Prentice Hall, 1997.

MEYROWITZ, N.; VAN DAM, A. Interactive editing systems: Part i. *ACM Computing Surveys*, v. 14, n. 3, p. 321–352, 1982.

MILLER, G. A. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, v. 63, n. 2, p. 81–87, 1956.

MURPHY, G. C.; LAI, A.; WALKER, R. J.; ROBILLARD, M. P. Separating features in source code: an exploratory study. In: *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, IEEE Computer Society, p. 275–284, 2001.

OSSHER, H.; TARR, P. *Multi-dimensional separation of concerns in hyperspace*. Relatório Técnico, IBM T. J. Watson Research Center, code: RC 21452(96717)16APR99, 1999.

OSSHER, H.; TARR, P. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, v. 44, n. 10, p. 43–50, 2001.

PACE, J. A. D.; CAMPO, M. R. Analyzing the role of aspects in software design. *Communications of the ACM*, v. 44, n. 10, p. 66–73, 2001.

PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, v. 15, n. 12, p. 1053–1058, 1972.

PERRELLA, R. Wiki: Home Page. Online, disponível em `http://texteditors.classichorseman.com/cgi-bin/wiki.pl?HomePage` - último acesso em 14/03/2005, 2005.

PESTOV, S. jEdit - Programmer's Text Editor. Online, disponível em `http://www.jedit.org/` - último acesso em 14/03/2005, 2005.

SEBESTA, R. W. *Concepts of Programming Languages*. 6 ed. Addison Wesley, 704 p., 2003.

SUN MICROSYSTEMS Overview (Java 2 Platform SE 5.0). Online, disponível em `http://java.sun.com/j2se/1.5.0/docs/api/` - último acesso em 14/03/2005, 2005.

TARR, P.; OSSHER, H. Hyper/J user and installation manual. available online at http://www.alphaworks.ibm.com/tech/hyperj, último acesso: 03/02/2005, 2000.

TARR, P.; OSSHER, H.; HARRISON, W.; STANLEY M. SUTTON, J. N degrees of separation: multi-dimensional separation of concerns. In: *ICSE '99: Proceedings of the 21st international conference on Software engineering*, p. 107–119, 1999.

THE ASPECTJ TEAM The AspectJ project at Eclipse.org. Online, disponível em `http://eclipse.org/aspectj/` - último acesso em 23/01/2006, 2006.

THE CME TEAM Concern Manipulation Environment. Online, available at `http://eclipse.org/cme/` - last access in 07/30/2005, 2005.

UNIX HELP Text editors. Online, disponível em `http://unixhelp.ed.ac.uk/editors/` - último acesso em 14/03/2005, 2005.

WEISS, D. M.; LAI, C. T. R. *Software product-line engineering: A family-based software development process*. Addison-Wesley, 1999.

Anexo A

Artigo "*Converting Theme-Oriented Design Modelling into a Fully Symmetric Approach*"

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo