Uma Arquitetura de Software baseada em Padrões de Projeto para o Desenvolvimento de Aplicações Concorrentes Confiáveis

Delano Medeiros Beder

Tese de Doutorado

Instituto de Computação
Universidade Estadual de Campinas

Uma Arquitetura de Software baseada em Padrões de Projeto para o Desenvolvimento de Aplicações Concorrentes Confiáveis

Delano Medeiros Beder

08 de Junho de 2001

Banca Examinadora:

- Cecília Mary Fischer Rubira (Orientadora)
- Avelino Francisco Zorzo
  Faculdade de Informática - PUCRS
- Eliane Martins
  Instituto de Computação - UNICAMP
- Luiz Eduardo Buzato
  Instituto de Computação - UNICAMP
- Taisy Silva Weber
  Instituto de Informática - UFRGS

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Beder, Delano Medeiros

B39a Uma arquitetura de software baseada em padrões de projeto para o desenvolvimento de aplicações concorrentes confiáveis / -- Campinas, [S.P. :s.n.], 2001.

Orientadora : Cecília Mary Fischer Rubira

Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Engenharia de software. 2. Tolerância a falhas (Computação). 3. Arquitetura de sistemas (Computação). 4. Padrões de projeto. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 08 de junho de 2001, pela Banca Examinadora composta pelos Professores Doutores:

Profa. Dra. Taisy Silva Weber
UFRGS

Prof. Dr. Avelino Francisco Zorzo
PUCRS

Prof. Dr. Eliane Martins
IC – UNICAMP

Prof. Dr. Luiz Eduardo Buzato
IC - UNICAMP

Profa. Dra. Cecília Mary Fischer Rubira
IC – UNICAMP

Uma Arquitetura de Software baseada em Padrões de Projeto para o Desenvolvimento de Aplicações Concorrentes Confiáveis

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Delano Medeiros Beder e aprovada pela Banca Examinadora.

Campinas, 08 de Junho de 2001.

Cecília Mary Fischer Rubira

Cecília Mary Fischer Rubira (Orientadora)

Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

Tinha eu 14 anos de idade quando meu pai me chamou
Perguntou-me se eu queria estudar filosofia
Medicina ou engenharia
Tinha eu que ser doutor
Mas a minha aspiração era ter um violão
Para me tornar sambista
Ele então me aconselhou:
Sambista não tem valor, nesta terra de doutor
E seu doutor, o meu pai tinha razão

Vejo um samba ser vendido, o sambista esquecido
O seu verdadeiro autor
Eu estou necessitado, mas meu samba encabulado
Eu não vendo não senhor!

(Paulinho da Viola)

Dedico este trabalho aos meus pais Zébédé e Clecy e às minhas irmãs Simone e Mônica pelo amor e incentivo dispensados ao longo de minha vida.

Agradecimentos

Este trabalho é resultado de muito esforço e perseverança dispensados nestes últimos anos. Quando escrevo estas palavras, é inevitável o sentimento de orgulho de que finalmente cheguei ao fim dessa longa jornada e de gratidão àqueles que contribuíram ao sucesso dessa jornada.
Gostaria de expressar minha sincera gratidão,

- à minha família pelo apoio incondicional e compreensão de que eu precisava buscar os meus objetivos longe de casa.
- à professora Cecília Rubira, pela sabedoria e coerência com que guiou esta jornada e pela paciência e amizade ao lidar com minhas dificuldades. Aproveito e agradeço também aos demais professores, funcionários e colegas do Instituto de Computação - UNICAMP pelos ensinamentos.
- To Alexander Romanovksy and Professor Brian Randell, for the discussions we had while designing the Station Case Study, and for the contributions and suggestions to my research.
- aos amigos que conheci e que sempre despertam meu respeito e admiração. A lista deles seria enorme; para não pecar pelo esquecimento, faço apenas referências gerais: amigos das inúmeras repúblicas em que morei; amigos da Pós-Graduação do Instituto de Computação - UNICAMP; demais amigos sempre presentes apesar da distância.
- ao CNPq e à CAPES pelo apoio financeiro.
- e, finalmente, a Deus por tudo que pedi e não recebi, pois o nada que eu ganhei era tudo que precisava.

— MUITO OBRIGADO !! —

Resumo

Sistemas computacionais complexos estão sujeitos a diferentes tipos de falhas, e a maneira mais adequada de lidar com tais falhas é aceitar que qualquer sistema pode apresenta-lás e empregar técnicas apropriadas para tolerá-las durante a execução do sistema. Desta forma, a abordagem mais apropriada para a construção de sistemas complexos confiáveis consiste na utilização de técnicas de tolerância a falhas que nos permitem definir regiões de confinamento e recuperação de erros. No entanto, técnicas de tolerância a falhas são geralmente utilizadas na fase de implementação do ciclo de desenvolvimento do sistema. Desta forma, não é freqüentemente fácil empregá-las, desde que projetistas necessitam levar em conta muitos detalhes de implementação.

Neste contexto, este trabalho faz duas contribuições relevantes. A primeira contribuição é a utilização prática de técnicas recentes de estruturação de sistemas na definição de uma arquitetura de software genérica para introduzir atomicidade, redundância de software, tratamento de exceções e recuperação de erros coordenada no desenvolvimento de sistemas orientados a objetos confiáveis durante o ciclo de desenvolvimento do sistema, iniciando-se na fase de projeto arquitetural passando pelo projeto detalhado e terminando na fase de implementação/codificação do sistema. A segunda contribuição é a definição de um conjunto coeso de padrões de projetos que refinam os elementos arquiteturais da arquitetura de software proposta e provêem uma clara e transparente separação de interesses entre a funcionalidade da aplicação e a funcionalidade relacionada à provisão da confiabilidade do sistema.

Abstract

Complex computer systems are prone to errors of many kinds, and the most reasonable way of dealing with them is to accept that any complex system has faults and to employ appropriate features for tolerating them during run time. We claim that the most beneficial way of achieving fault tolerance in complex systems is to use system structuring which has fault tolerance measures associated with it. In this case, structuring units serve as natural areas of error containment and error recovery. However, these techniques are mainly developed for employment at the implementation phase of the system development. Hence, it is not often easy to apply them correctly, as the designers have to take into account many implementation details.

In this context, this work makes two main contributions. The first contribution is the practical employment of recent system structuring techniques in the definition of a generic software architecture for introducing atomicity, exception handling, and coordinated error recovery into dependable object-oriented systems at the earlier phases of system development. That is, from architectural design, through detailed design to coding. The second contribution is the definition of a set of design patterns which refine the architectural elements of the proposed software architecture and provide a clear and transparent separation of concerns between the application functionality, and the functionality related to providing system dependability.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

O uso crescente de sistemas computacionais em quase todos os ramos da sociedade tem levado a necessidade de desenvolvimento de sistemas confiáveis. Por exemplo, considere um banco que tem os seus serviços automatizados interrompidos devido à falhas em sistemas computacionais, então perdas financeiras podem ocorrer, com conseqüências desastrosas. Portanto, confiabilidade é um requisito importante para o desenvolvimento de sistemas computacionais.

Sistemas complexos estão sujeitos a diferentes tipos de falhas, e a forma mais adequada de lidar com tais falhas é empregar técnicas apropriadas para tolerá-las durante a execução do sistema. Nós acreditamos que a abordagem mais apropriada para a construção de sistemas complexos confiáveis é empregar técnicas de estruturação de software, com propriedades de tolerância a falhas associadas, que nos permitem definir regiões de confinamento e recuperação de erros. Tratamento de exceções provê um esquema adequado para detectar e tratar erros e também incorporar atividades de tolerância a falhas em sistemas de software. A detecção de um erro resultará na sinalização de uma exceção e na invocação de um tratador de exceção que implementa as atividades de recuperação de erros.

Muitos sistemas computacionais modernos são concorrentes. Exceções são mais difíceis de serem tratadas em sistemas concorrentes em que componentes cooperam para realizar uma atividade em conjunto. Neste contexto, erros podem ser propagados através da comunicação intercomponentes. A noção de ação atômica [74, 97] é de importância vital na estruturação das atividades e no emprego de tratamento de exceções em sistemas concorrentes. Usando esta abordagem, projetistas podem estruturar suas atividades concorrentes como uma coleção de ações atômicas que servem como regiões de confinamento e recuperação de erros. Recentemente, o conceito de ações atômicas coordenadas [130], que estende a noção de ação atômica, foi introduzido que permite desenvolvedores projetar, estruturar e prover tolerância a falhas usando diferentes técnicas (incluindo, tratamento

de exceções e diversidade de projeto) em sistemas concorrentes.

Problema

No entanto, existem vários fatores que complicam o uso de técnicas de tolerância a falhas durante o desenvolvimento de sistemas: (i) linguagens de programação não dão suporte ao emprego delas diretamente; (ii) estas técnicas são principalmente empregadas na fase de implementação do ciclo de desenvolvimento do sistema; (iii) não é fácil empregá-las corretamente, posto que os projetistas tem que levar em conta muitos detalhes de implementação.

1.1 Objetivos

Os principais objetivos desse trabalho são:

- Projeto e implementação de requisitos de tolerância a falhas e sua incorporação explícita e bem estruturada na descrição de uma arquitetura de software [108].
- Utilização prática de técnicas recentes de estruturação de software, com características de tolerância a falhas associadas, desde as fases iniciais do ciclo de desenvolvimento de sistemas orientado a objetos confiáveis, isto é, iniciando no projeto arquitetural, passando pelo projeto detalhado e terminando na codificação/implementação do sistema.

1.2 Solução proposta

Nós propomos uma arquitetura de software para o desenvolvimento de sistemas concorrentes confiáveis baseado em três noções: (i) componente tolerante a falhas ideal [74]; (ii) colaborações entre papéis de componentes [128] e (iii) reflexão computacional [84].

A arquitetura de software proposta define uma estrutura em camadas onde cada camada provê serviços para a camada superior e usa os serviços da camada inferior na hierarquia. Em adição, componentes tolerante a falhas ideais presentes em uma mesma camada podem realizar atividades cooperativas (isto é, colaborações), sendo que cada componente tolerante a falhas ideal desempenha um papel de componente diferente em cada atividade cooperativa.

Reflexão computacional define uma arquitetura de níveis, composta de um nível base onde a funcionalidade da aplicação é implementada e um meta-nível onde meta-componentes são responsáveis pela implementação de propriedades não-funcionais da aplicação, tais como distribuição e tolerância a falhas, de forma transparente para projetistas de aplicações. Projetistas podem empregar a noção de separação de interesses[1] e se concentrar na definição das propriedades funcionais da aplicação, abstraindo-se assim das suas propriedades não-funcionais.

Ademais, nós propomos um conjunto coeso de padrões de projeto que refinam e provêem o projeto detalhado dos componentes e conectores da arquitetura de software proposta. Estes padrões de projeto provêem soluções de projeto para implementar técnicas de tolerância a falhas, a saber, tratamento de exceções, detecção de erros, recuperação de erros coordenada e redundância de software. Nós utilizamos a linguagem de programação Java [48] e um protocolo de meta-objetos chamado Guaraná [93] na construção de um *framework* orientado a objetos que provê uma infra-estrutura genérica para o desenvolvimento de sistemas concorrentes confiáveis.

1.3 Contribuições

Este trabalho apresenta as seguintes contribuições:

- Uma arquitetura de software genérica para introduzir atomicidade, redundância de software, tratamento de exceções e recuperação de erros coordenada no desenvolvimento de sistemas orientados a objetos confiáveis desde as fases iniciais do ciclo de desenvolvimento do sistema.

- Um conjunto coeso de padrões de projetos que refinam os componentes lógicos e os conectores da arquitetura de software proposta e provêem uma clara e transparente separação de interesses entre a funcionalidade da aplicação e a funcionalidade relacionada à provisão da confiabilidade do sistema.

- Um *framework* orientado a objetos que provê uma infra-estrutura genérica para o desenvolvimento de sistemas concorrentes confiáveis. O conjunto de padrões que refinam os componentes lógicos e conectores elementos da arquitetura de software proposta foram utilizados na definição deste *framework*.

- Uma abordagem sistemática na incorporação de características de tolerância a falhas durante o ciclo de desenvolvimento do sistema, iniciando-se na fase de projeto

[1] Em inglês: *separation of concerns*.

arquitetural passando pelo projeto detalhado e terminando na fase de codificação do sistema. Esta abordagem identifica decisões de projeto adequadas para construir sistemas orientados a objetos confiáveis e apresenta um conjunto de diretrizes que deveria ser seguido durante todo o desenvolvimento do sistema.

- Projeto e implementação de três estudos de casos que ilustram a aplicabilidade da arquitetura de software proposta.

1.4 Trabalhos relacionados

- **Tratamento de exceções em linguagens de programação.** Uma das principais deficiências dos mecanismos de tratamento de exceções disponíveis é a inexistência de suporte apropriado para o tratamento de exceções concorrentes. Os mecanismos existentes geralmente são dedicados para programas seqüenciais. Somente a linguagem Arche [64] provê um esquema para tratamento de condições excepcionais em sistemas concorrentes. Entretanto, o modelo de concorrência implementado em Arche limita-se a ativar recuperação de erros entre objetos do mesmo tipo. Assim, as linguagens de programação orientadas a objetos atuais não provêem, de forma satisfatória, mecanismos de tratamento de exceções adequados para o desenvolvimento de sistemas orientado a objetos confiáveis.

- **Tolerância a falhas em sistemas concorrentes.** Recentemente, alguns trabalhos [101, 135, 137] têm proposto mecanismos dedicados especialmente para tratamento de exceções concorrentes como extensões para determinadas linguagens de programação específicas. As abordagens propostas exigem alguma modificação da linguagem de programação e/ou de seu compilador ou interpretador, ou a definição de uma interface de programação para tratamento de exceções concorrentes.

  Zorzo *et al.* [137] desenvolveu um *framework* orientado a objetos para implementar o conceito de ações atômicas coordenadas que provê aos projetistas um conjunto de classes que auxilia a estruturação de suas aplicações concorrentes. Entretanto, a abordagem proposta por este trabalho é intrusiva sobre o ponto de vista da aplicação uma vez que o código da aplicação possui uma série de chamadas a serviços específicos do mecanismo de tratamento de exceções concorrentes. O código extra inserido dificulta a legibilidade, a reutilização e manutenção dos componentes da aplicação.

  Zorzo [135] propôs um *framework* orientado a objetos para implementar *Dependable Multiparty Interactions (DMI)*. Cada *DMI* é um grupo de interações entre múltiplos participantes: uma interação básica que implementa a atividade cooperativa dos

participantes e diversas interações que tratam as exceções que podem ser levantadas durante a execução da *DMI* (ou durante a execução da interação básica ou durante uma das interações que tratam as exceções). Utilizando a abordagem proposta por este trabalho, *DMIs* são construídas através do encadeamento de interações não confiáveis, onde cada interação na cadeia implementa os tratadores para as exceções levantadas na interação anterior na cadeia. Apesar desta abordagem apresentar uma separação explícita entre o código normal e excepcional da aplicação, os mecanismos de tratamento de exceções ainda não são totalmente transparentes sob o ponto de vista da aplicação. Finalmente, estes dois trabalhos são dependentes de linguagem de programação e do mecanismo de tratamento de exceções utilizado.

- **Arquiteturas reflexivas para o desenvolvimento de sistemas confiáveis.** Alguns trabalhos [43, 50] têm sido propostos que auxiliam o desenvolvimento de sistemas orientados a objetos confiáveis. FRIENDS [43] é uma arquitetura reflexiva que provê uma biblioteca de meta-objetos para tolerância a falhas, comunicação segura e distribuição. GARF [50] é um ambiente de programação distribuída que permite ao projetista da aplicação inicialmente projetá-la em um ambiente centralizado e então distribuir seus componentes e aumentar a confiabilidade da aplicação por replicar os componentes críticos da aplicação. Tais trabalhos, no entanto, lidam com a provisão de características de tolerância a falhas apenas na fase de implementação do ciclo de desenvolvimento do sistema. Nós argumentamos que melhores resultados poderiam ser alcançados se técnicas de tolerância a falhas fossem incorporadas de forma sistemática e disciplinada durante todo o ciclo de desenvolvimento do sistema. Além disso, tais trabalhos não apresentam suporte ao tratamento de falhas de software através do projeto de componentes de software utilizando diversidade de projeto [4, 74, 97].

- **Arquiteturas de software para o desenvolvimento de projetos baseados em colaborações.** Lemos [36] propôs um estilo arquitetural que é utilizado na representação de arquiteturas de sistemas que utilizam o projeto baseado em colaborações como base. Na abordagem proposta por este trabalho, colaborações também são conectores que descrevem a interação entre componentes. Entretanto, tal abordagem não apresenta nenhum suporte ao desenvolvimento de atividades cooperativas confiáveis.

1.5 Organização do texto

O restante deste texto está organizado da seguinte forma:

Capítulo 2 apresenta os conceitos fundamentais sobre tolerância a falhas com intuito de facilitar a compreensão dos próximos capítulos.

Capítulo 3 introduz as técnicas de engenharia de software utilizadas no desenvolvimento deste trabalho.

Capítulo 4 apresenta a arquitetura de software proposta baseada em três noções: (i) componente tolerante a falhas ideal, (ii) colaborações entre papéis de componentes e (iii) reflexão computacional.

Capítulo 5 apresenta um conjunto coeso de padrões de projetos que refinam os elementos arquiteturais da arquitetura de software proposta e provêem uma clara e transparente separação de interesses entre a funcionalidade da aplicação e a funcionalidade relacionada à provisão da confiabilidade do sistema.

Capítulo 6 apresenta uma abordagem sistemática na incorporação de características de tolerância a falhas durante o ciclo de desenvolvimento do sistema e propõe um conjunto de diretrizes que deveria ser seguido durante o desenvolvimento do sistema.

Capítulos 7 e 8 ilustram a aplicabilidade da arquitetura de software proposta através do projeto e implementação de 3 estudos de casos.

Capítulo 9 resume as conclusões deste trabalho, apresentando as principais contribuições e os possíveis trabalhos futuros.

Capítulo 2

Fundamentos de tolerância a falhas

Este capítulo traz uma compilação dos conceitos empregados na teoria e prática de tolerância a falhas. Uma discussão completa dos conceitos desenvolvidos na área de tolerância a falhas está fora do escopo deste trabalho. Limitar-nos-emos, portanto, a apresentar os conceitos e técnicas de tolerância a falhas utilizados no desenvolvimento deste trabalho. Os trabalhos de Heimerdinger e Weisnstock [59], Jalote [66], Lee e Anderson [74], Lisbôa e Azeredo [77], Laprie [73], Lyu e Krishnamurthy [82] e Lyu [81] podem ser consultados para complementar o que será exposto neste capítulo.

2.1 Definições básicas

Antes de discutirmos os aspectos encontrados nos sistemas tolerantes a falhas, devemos definir alguns conceitos básicos. Nesta seção apresentaremos os conceitos básicos de tolerância a falhas baseado nos seguintes trabalhos [59, 66, 74, 77].

2.1.1 Sistema computacional

Antes de discutirmos o que são sistemas tolerantes a falhas, devemos definir o que é um sistema.

- Um sistema é definido como um conjunto de sub-sistemas ou componentes que interagem sob o controle de um projeto [74].

Um componente de um sistema, por sua vez, é um outro sistema, com seu próprio comportamento e sua própria estrutura interna e portanto poderá também ter subcomponentes (Figura 2.1). O projeto de um sistema também é um componente, mas

com características especiais, como exemplo, a responsabilidade de controlar a interação entre os componentes do sistema. Como resultado da atividade do sistema, o componente receberá requisições para prover um serviço. No intuito de atender estas requisições, componentes possivelmente farão requisições de serviços para seus próprios sub-componentes. Supondo que estes sub-componentes atendem estas requisições de forma satisfatória, eles fornecem respostas para o componente, que por sua vez irá responder à requisição de serviço do sistema.

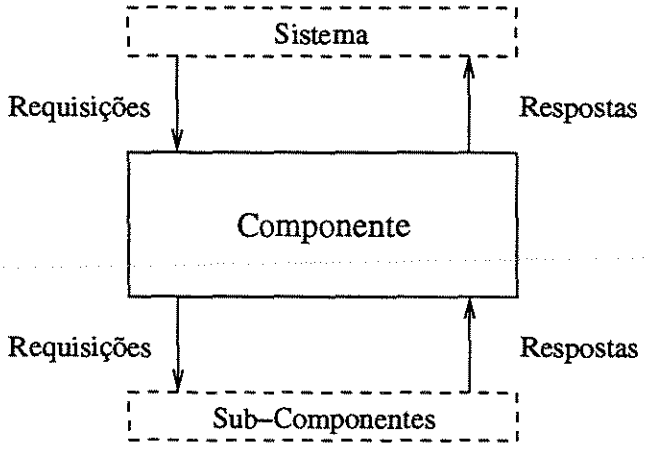


Figura 2.1: Componente geral de um sistema

O conceito de componente usado é geral e pode ser implementado em hardware ou software. Como caso especial (e para evitar recursão infinita) um componente pode ser considerado atômico, com a implicação de que a sua estrutura interna não precisa ser determinada e portanto pode ser abstraída.

O comportamento externo do sistema pode ser considerado como uma abstração de seu estado interno. O estado interno de um sistema compreende os estados internos de seus componentes. Durante a sua execução, o estado interno do sistema constantemente é alterado como conseqüência da interação entre seus componentes. Fica claro que o padrão de comportamento da interação entre os componentes deve ser estabelecido e controlado. Este é o propósito do projeto do sistema. O projeto controla quais componentes interagem e como, e também determina o modo em que interações entre o sistema e seu ambiente influenciam o comportamento dos componentes.

2.1.2 Falhas, erros e defeitos

Os termos falha, erro e defeito[1] devem ser definidos cuidadosamente no contexto de tolerância a falhas porque são muitas vezes considerados sinônimos na escrita leiga. O importante é a relação falha → erro → defeito. Quando há uma falha no sistema, sua manifestação dá origem a erros, os quais possivelmente resultam em um subseqüente defeito no sistema.

O comportamento externo de um sistema é percebido pelos usuários como a alternância entre dois estados do serviço fornecido em relação ao serviço especificado:

- serviço normal: o serviço é fornecido como especificado;
- serviço anormal: o serviço fornecido é diferente do serviço especificado.

Em vista do importante papel desempenhado pela especificação, é apropriado determinar as características desejáveis de uma especificação. Uma especificação deve ser consistente, completa e autoritária, e pode ser aplicada como um teste eficaz em todas as circunstâncias para determinar se o comportamento do sistema está ou não aceitável. Sob este ponto de vista, é razoável partir do princípio que a especificação está correta e que o projeto e a implementação do sistema devem ser consistentes com esta especificação.

Desta forma, dada uma especificação de um sistema, o conceito de defeito pode ser definido:

- Um defeito em um sistema ocorre quando o comportamento do sistema desvia do especificado [74].

Isto é, o sistema não forneceu a resposta correta, ou forneceu a resposta correta mas não atendeu à restrição de tempo especificada ou mesmo interrompeu anormalmente a sua execução.

Uma vez que a especificação é autoritária e não pode ser modificada, então as causas de um defeito estão apenas relacionadas à própria atividade interna do sistema. E isto nos leva a definição de dois conceitos que são essenciais na discussão das causas de um defeito: os conceitos de transição errônea e de estado errôneo.

Como descrito anteriormente, a atividade de um sistema é definida como a seqüência de transições do estado externo daquele sistema e o comportamento do estado externo é uma abstração de seu estado interno. Supondo que, durante a sua atividade, o sistema percorre os seguintes estados s_1, s_2, s_3, ..., s_n. E que, entre os estados s_1 e s_{n-1}, seu comportamento atende a especificação do sistema, porém, ao entrar no estado s_n, o seu comportamento

[1] Em inglês: *fault, error e failure*, respectivamente.

externo é diferente do especificado. Uma vez que o sistema foi especificado para não falhar, em algum ponto na seqüência s_1, ... , s_n houve um desvio do comportamento especificado. Supondo que este desvio aconteceu quando o sistema passou para o estado s_i. A transição entre s_{i-1} e s_i deveria então ser considerada responsável pela eventual falha do sistema e poderia conseqüentemente ser considerada como errônea. Neste caso, os estados s_1, ... , s_{i-1} são válidos, porém todos os demais estados entre s_i e s_n são errôneos, pois o sistema não deveria ter passado por estes estados. Intuitivamente, uma transição errônea põe o sistema em um estado interno no qual um defeito pode acontecer. Entretanto, existe a possibilidade de que este defeito não venha a ocorrer. Isto é, o estado errôneo pode ser detectado, e ações corretivas podem ser empregadas no intuito de levar o sistema para um estado válido antes do defeito ocorrer.

Uma vez que uma transição errônea foi identificada como a causa de um possível defeito, é natural perguntar o que causou a transição errônea. Transições de estados internos são determinadas pelos componentes do sistema em função do seu projeto. Se um ou mais componentes não atendem às suas especificações, então isto poderia levar o sistema a um estado errôneo. Entretanto, se todos componentes atendem as suas especificações, então o problema deve estar no projeto do sistema. Se existe uma transição errônea e nenhum componente falhou, então o projeto do sistema deve ter falhado. Desta forma, uma transição errônea deve ser conseqüência de defeito ou em um componente ou no projeto do sistema.

Quando um sistema está em um estado errôneo, a avaliação dos estados externos dos componentes do sistema permite que seja feita a decisão sobre quais componentes terão que alterar seus estados externos no intuito de tornar o estado interno do sistema válido. Os estados de tais componentes são denominados erros no sistema. Desta forma, o conceito de erro pode ser definido:

- Um erro é a parte do estado errôneo que constitui a diferença em relação ao estado válido [74].

Como já discutido, se um sistema está em um estado errôneo, então um defeito no projeto ou em um dos seus componentes ocorreu. Desta forma, erros em componentes ou no projeto podem levar a erros no sistema. No intuito de distinguir erros em diferentes níveis no sistema e para facilitar discussão dos relacionamentos de causa e efeito, a definição de falha será apresentada:

- Um erro em um componente ou no projeto do sistema será denominado como uma falha [74]:
  - Uma falha no componente de um sistema causa um erro no estado interno de um componente.

– Uma falha no projeto de um sistema é um erro no estado do projeto.

Uma falha pode levar o sistema a apresentar um erro que poderá finalmente manifestar-se como um defeito. Note entretanto que a única diferença entre uma falha e um erro está relacionada com a estrutura do sistema. Uma falha em um sistema é um erro em um componente ou no projeto do sistema.

2.1.3 Componente ideal tolerante a falhas

Como já discutido, um sistema consiste de um conjunto de componentes que interagem sob o controle de um projeto. Componentes recebem requisições de serviços e produzem respostas a estas requisições (Figura 2.1). Com o intuito de produzir respostas a uma dada requisição de serviço, um componente pode solicitar a seus sub-componentes um determinado serviço.

Desta forma, podemos considerar um sistema como uma hierarquia de componentes, cada componente provendo algum serviço de acordo com alguma especificação. Esta hierarquia pode ser representada como um grafo acíclico, em que componentes são representados por vértices, e uma aresta de um vértice A para o vértice B significa que o componente A é usuário dos serviços provido pelo componente B, e o completo sucesso de A depende do completo sucesso de B.

As respostas de um componente podem ser classificadas em duas categorias: normal e anormal. Respostas normais são aquelas que um componente produz quando tudo ocorre como projetado e as respostas anormais são aquelas produzidas quando o comportamento do componente se desvia do projetado. As respostas anormais de um componente são denominadas exceções e significam que alguma situação excepcional ocorreu neste componente ou em algum sub-componente deste.

Como conseqüência do particionamento das respostas em duas categorias, a atividade de um componente também pode ser particionada em atividade normal e atividade anormal (tratamento de exceções). A atividade normal implementa o serviço especificado para o componente enquanto a atividade anormal implementa as medidas para tolerar as falhas causadas pelo desvio do comportamento previsto. Esta separação entre a atividade normal e anormal leva-nos à definição do componente tolerante a falhas ideal [74] (Figura 2.2).

Exceções podem ser classificadas em três grupos: exceções de interface, exceções internas e exceções de defeito. Exceções de interface são sinalizadas em resposta a uma requisição de um serviço não disponível na interface do componente. Exceções internas são geradas quando um componente detecta uma situação inesperada durante sua atividade normal. Se esta exceção não pode ser tratada pelo próprio componente, então

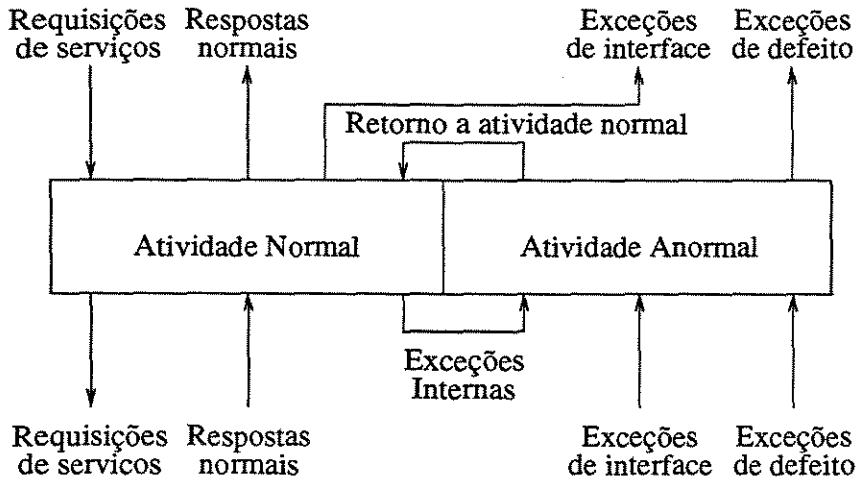


Figura 2.2: Descrição de um componente tolerante a falhas ideal

uma exceção de defeito é levantada indicando que, por alguma razão, este componente não pode prover o serviço especificado. Se um componente recebe uma resposta anormal (exceção de interface ou de defeito) de uma invocação de um outro componente ou detecta uma condição anormal (exceção interna) durante sua execução normal, ele invoca os mecanismos apropriados de tolerância a falhas. Se estas exceções são tratadas, então o componente pode voltar a prover o serviço normal. Entretanto, se o componente não consegue tratar a exceção, uma exceção de defeito é sinalizada.

2.1.4 Redundância

Técnicas de obtenção de tolerância a falhas são baseadas em redundância. Redundância de hardware é obtida através da replicação de componentes. A idéia é ter réplicas de componentes críticos de hardware de forma que, se algum deles falhar, existirão réplicas para substituí-los. A replicação de componentes de hardware é usada efetivamente porque o projeto do sistema de hardware é suposto correto, e o objetivo então é tratar algumas falhas que ocorrem devido a razões físicas. Um bom exemplo do uso de redundância para tolerar falhas em componentes de hardware é a Redundância Modular Tripla[2] que é utilizada para evitar a degradação do desempenho do sistema na presença de falhas. Desta forma, para tolerar uma falha no componente A (Figura 2.3(a)), o componente A poderia ser substituído pelo RMT ilustrado na Figura 2.3(b). O RMT consiste de 3 cópias idênticas do componente A (projeto idêntico) e um componente de votação V (implementando a detecção de erros) o qual verifica as respostas dos componentes e seleciona a resposta dada pela maioria. O sistema ilustrado nesta figura é projetado para tolerar falhas em

[2]Em inglês: *Triple Modular Redundancy* (TMR).

qualquer cópia do componente A, por apenas produzir respostas nas quais pelo menos dois componentes concordam. Uma generalização desta técnica é a Redundância N-Modular[3] (RNM) em que o componente físico é replicado N vezes.

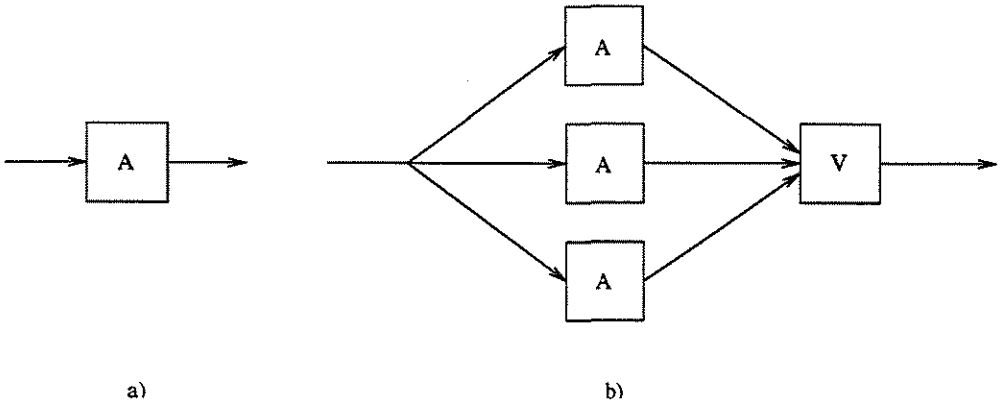


Figura 2.3: Redundância modular tripla

Em software, a replicação de componentes com projetos idênticos não é suficiente. Caso um componente de software tenha uma falha de projeto, então todas as réplicas do componente terão a mesma falha de projeto. Por esse motivo, diversidade de projeto é de fundamental importância e implica que componentes de projetos distintos são replicados. As várias técnicas de tolerância a falhas de software podem ser divididas em duas categorias: redundância dinâmica e redundância estática [99].

Um sistema com redundância dinâmica consiste de diversos componentes redundantes com a restrição que apenas um componente está ativo. Se um erro é detectado, o componente ativo é substituído por um outro componente sobressalente. Um exemplo do uso da redundância dinâmica é o bloco de recuperação [97] (Seção 2.4.1).

A redundância estática utiliza componentes extras de software de tal forma que os efeitos de um ou mais erros são mascarados e não percebidos pelos usuários do sistema. Um bom exemplo do uso da redundância estática é técnica de N-versões [4] (Seção 2.4.2).

2.2 Fases no processamento de falhas

A implementação de mecanismos efetivos para a tolerância a falhas depende da arquitetura e do projeto de um sistema. Conseqüentemente, não existem técnicas gerais para a inclusão de mecanismos de tolerância a falhas em sistemas. Contudo, nota-se que os sistemas tolerantes a falhas incluem mecanismos que colocam o sistema em uma de quatro

[3]Em inglês: *N-Modular Redundancy* (NMR).

fases durante o processamento de uma falha: (i) detecção de erros, (ii) confinamento e avaliação dos danos, (iii) recuperação de erros e (iv) tratamento de falhas e continuidade do serviço.

2.2.1 Mecanismos de detecção de erros

O ponto inicial de qualquer atividade de tolerância a falhas é a detecção de erros. Como já discutido, falhas e defeitos não podem ser diretamente observados, porém podem ser deduzidas através da presença de erros. Desde que erros são definidos pelo estado do sistema, mecanismos de detecção de erros podem ser introduzidos para verificar se existe um erro ou não. Idealmente, o mecanismo de detecção de erros deveria detectar todo erro causado por falhas que o esquema de tolerância a falhas pretende tolerar. Isto nos leva a definição de mecanismos ideais de verificação.

Verificação ideal. Considere um sistema S que foi projetado para prover um certo serviço. Se o comportamento de S não atende à especificação, então S falhou. Uma vez que o propósito de tolerância a falhas é prevenir defeitos no sistema, uma importante técnica para detecção de erros pode ser baseado na interceptação das respostas produzidas pelo sistema e verificar se aquelas respostas de fato atendem à especificação. Isto resulta em um novo sistema S' composto por S e por um componente extra que realiza a verificação, como ilustrado na Figura 2.4.

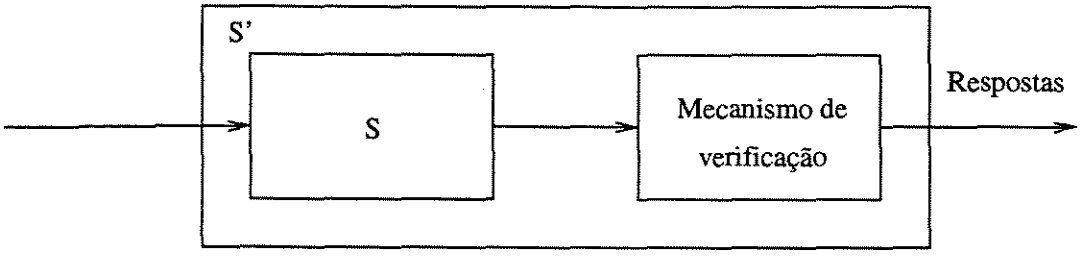


Figura 2.4: Sistema com mecanismo ideal de verificação

Existem algumas propriedades que um mecanismo ideal de verificação deve satisfazer [74]:

1. Um mecanismo ideal de verificação deve ser determinado unicamente pela especificação do sistema e não deve ser influenciado pelo projeto interno do sistema; Conseqüentemente, para propósitos de projeto de um mecanismo de verificação, o sistema deve ser tratado como uma caixa preta [66].

2. Um mecanismo ideal de verificação deve ser completo e correto. Isto implica que o mecanismo deve ser apto para detectar todos os possíveis erros no comportamento do sistema e que ele nunca deve declarar um erro que não existe.

3. O mecanismo de verificação deve ser independente do sistema em relação a suscetibilidade de falhas. Se o mecanismo de verificação falha no mesmo momento que o sistema falha, então este mecanismo não tem nenhum valor prático.

Na prática, esses critérios rigorosos não podem ser materializados em muitos sistemas. A materialização de um mecanismo completo de verificação geralmente não é muito viável, pois tal mecanismo pode ser muito complexo e conseqüentemente sujeito a falhas. Além disso, a independência entre o sistema e seu mecanismo de verificação não pode ser absoluto, por duas razões. Primeiro, o mecanismo de verificação obviamente terá acesso à informação a ser verificada, e pode, desta forma, corromper aquela informação. Segundo, a implementação de qualquer mecanismo de verificação será baseada em suposições sobre a estrutura do sistema e conseqüentemente sobre o tipo de falhas que podem ocorrer.

Em vista destas limitações, mecanismos de verificação aceitáveis são freqüentemente empregados para detecção de erros. Um mecanismo de verificação aceitável não é um mecanismo ideal de verificação, mas uma aproximação deste. O objetivo dos mecanismos de verificação aceitáveis é manter o custo baixo e ao mesmo tempo maximizar o número de erros detectados. Tais mecanismos não garantem que todos erros serão detectados, porém, tentam detectar uma grande parte dos erros, particularmente aqueles que estão mais propensos a ocorrer.

Tais mecanismos de verificação podem ser de diferentes tipos, dependendo do sistema e das falhas de interesse. Entretanto, existem alguns mecanismos de verificação que são freqüentemente usados.

Mecanismos baseados em replicação. São uns dos mais comuns e poderosos mecanismos de verificação. Mecanismos baseados em replicação podem ser empregados sem o conhecimento da estrutura interna do sistema. Como o nome sugere, tal mecanismo envolve replicar alguns componentes do sistema. Os resultados dos diferentes componentes são comparados, ou votados, para detectar erros.

Mecanismos baseados em contratos. Um contrato define os aspectos observáveis pelos clientes de um componente. Contratos são definidos através de precondições, póscondições e invariantes. Se uma destas condições é violada, então um erro é detectado. Um contrato especifica quais são as restrições sobre os parâmetros de entrada de um serviço que devem ser satisfeitas pelo cliente, ou seja, as precondições, e qual é o comportamento que pode ser esperado pelo cliente na execução de um serviço, caso estas restrições sejam

atendidas, ou seja as pós-condições. Além disso, especifica as invariantes do componente, que são restrições sobre o estado interno do componente.

2.2.2 Mecanismos de confinamento e avaliação dos danos

Ao se detectar um erro em um sistema, sabe-se que em algum ponto do sistema uma falha está presente. Entretanto, pode existir um atraso entre a manifestação da falha e a detecção dos erros. Devido as interações entre componentes do sistema durante este período, um erro pode propagar e se espalhar para os outros componentes do sistema. Desta forma, após a detecção dos erros e antes da sua correção, é necessário determinar as fronteiras do dano, isto é, a sua extensão. Este é o objetivo da fase de confinamento e avaliação dos danos.

Erros se propagam por intermédio da comunicação entre os componentes do sistema. Conseqüentemente, a determinação da extensão dos danos causados por erros depende do exame do fluxo de informação entre diferentes componentes. O exame do fluxo de informação entre componentes do sistema pode revelar as fronteiras que confinam o erro.

A determinação das fronteiras de um erro pode ser facilitada pelo uso de mecanismos que organizam as atividades (computação e comunicação) do sistema em fluxos bem definidos. Neste contexto, o conceito de ação atômica [74] torna-se importante. O suporte a ações atômicas deve permitir que um componente, entre, execute operações sob o controle do mecanismo de ação atômica e então saia da ação atômica. Uma ação não será atômica se, antes de seu término, ocorrer um fluxo de informação de ou para componentes externos àquela ação.

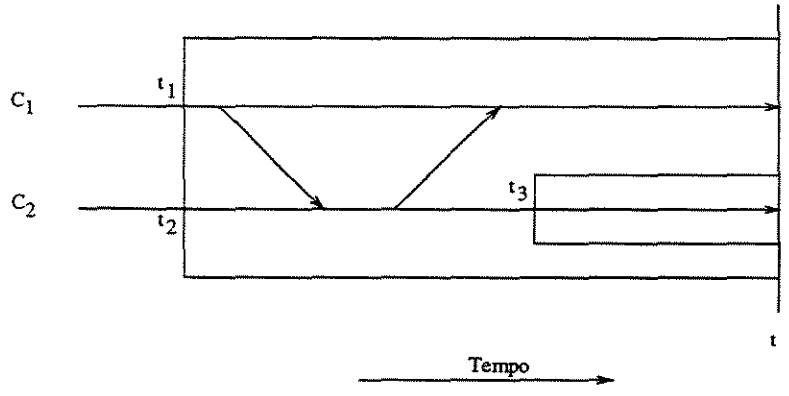


Figura 2.5: Ações atômicas

Por exemplo, a Figura 2.5 ilustra dois componentes que participam de uma ação atômica (as ações atômicas são representados pelas linhas retangulares). Se o componente C_2 detectou um erro no instante t, então uma estratégia para a avaliação dos danos deveria

supor que todas suas atividades desde t_3 estão suspeitas e que as mudanças em seu estado deveriam ser verificadas (ou abandonadas). Levando em conta que a atividade de C_2 desde t_3 tem sido atômica, o erro não poderia ter sido propagado para C_1. Analogamente, se C_1 detectou um erro no instante t, então todas suas atividades desde t_1 deveriam ser consideradas suspeitas. Devido a interações entre os componentes C_1 e C_2, todas as atividades de C_2 desde t_2 também deveriam ser consideradas suspeitas e portanto deveriam ser verificadas (ou abandonadas).

2.2.3 Mecanismos de recuperação de erros

Uma vez que o erro foi detectado e a sua extensão identificada, é necessário remover este erro do sistema. Nesta fase, o sistema é posto em um estado livre de erros, isto é, o sistema é posto em um estado consistente. Há dois mecanismos básicos para a recuperação de erros: recuperação de erros por retrocesso de estado que consiste na passagem do estado do sistema para um estado consistente já ocorrido e recuperação de erros por avanço de estado que consiste na passagem do estado do sistema para um estado consistente ainda não ocorrido.

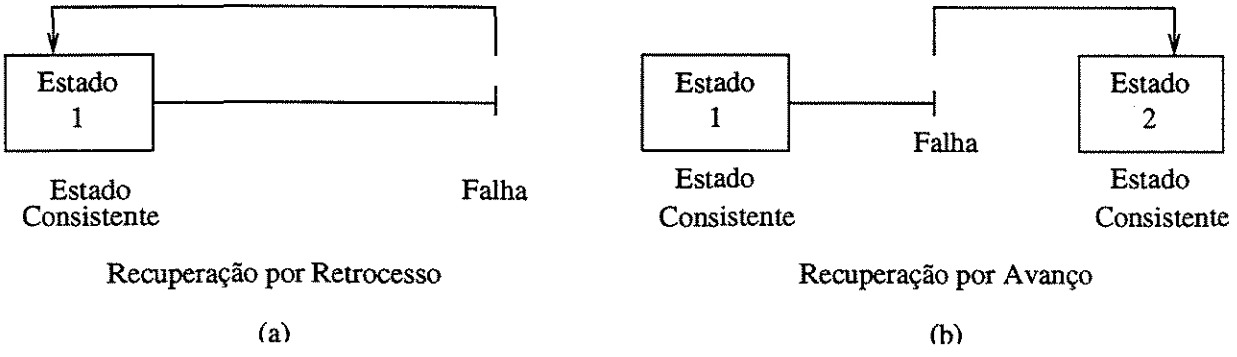


Figura 2.6: Mecanismos de recuperação de erros

A distinção entre estes dois mecanismos origina-se da possibilidade de projetistas do sistema preverem algumas das falhas que podem ocorrer durante a execução do sistema. Para as falhas previstas, é possível implementar uma estratégia de recuperação de erros que consiste em fazer mudanças específicas e limitadas no estado do sistema, promovendo-o a um estado consistente futuro. Assim, a recuperação de erros ocorre através do avanço de seu estado (Figura 2.6(b)). Este mecanismo é denominado recuperação de erros por avanço. Exceções e tratamento de exceções são os mecanismos comuns para prover recuperação por avanço [30]. Para a situação de falhas não previstas pelos projetistas, a única solução viável é substituir o completo estado do sistema. Este mecanismo restaura o sistema ao estado anterior à manifestação da falha e é denominado recuperação de erros

por retrocesso (Figura 2.6(a)). Os mecanismos de recuperação por retrocesso dependem da provisão de pontos de recuperação[4], isto é, um meio pelo qual o estado do sistema possa ser armazenado e mais tarde restaurado [17, 63, 69].

A recuperação de estados errôneos por retrocesso é uma das formas mais empregadas. É de natureza geral, e não depende muito da natureza da falha ou defeito. No entanto, existem duas questões que devem ser discutidas. Primeiro, neste mecanismo necessita-se que pontos de recuperação sejam periodicamente estabelecidos. Isto afeta a execução normal do sistema mesmo que falhas não aconteçam. Ao contrário, mecanismos de recuperação por avanço são normalmente mais econômicas, pois mudanças são apenas realizadas naqueles componentes que são suspeitos de estarem errôneos. Segundo, existem aplicações em que apenas o uso do mecanismo de recuperação com retrocesso não é o mais apropriado. Aplicações que provocam um efeito no mundo externo (por exemplo, mensagens impressas no vídeo) não podem ser restaurados ao seu estado anterior. Para tais sistemas apenas o uso do mecanismo de recuperação de erros com retrocesso não é adequado. Destes exemplos podemos afirmar que uma solução abrangente deve combinar os dois mecanismos. Esta solução poderá ser usada por um grupo maior de aplicações que poderão selecionar os mecanismos de recuperação que melhor a servem.

2.2.4 Tratamento de falhas e continuação do serviço

Nas três primeiras fases, o foco de atenção está na descoberta de erros. Ele é detectado, confinado e removido. Estas ações são suficientes para restabelecer a operação normal do sistema no caso de erros causados por falhas transientes. Entretanto, se as falhas são permanentes, então a falha que causou o erro e possivelmente o defeito, ainda permanece no sistema, mesmo após a recuperação de erros. Neste caso, se reexecutarmos o sistema, a mesma falha produzirá os mesmos erros. Para evitar que isto aconteça, é essencial que os componentes falhos sejam identificados e não utilizados, ou utilizados em uma nova configuração, na computação realizada após a recuperação de erros. Este é o objetivo da fase de tratamento de falhas.

Esta fase constitui de duas sub-fases: localização da falha e conserto do sistema. Na fase de localização da falha, o componente que contem a falha é identificado. Na fase de conserto do sistema, o sistema é "reparado" de tal forma que componentes falhos não são usados ou usados em uma configuração diferente. Uma estratégia simples é substituir os componentes falhos por componentes redundantes. Tais componentes são projetados no intuito de implementar a mesma especificação dos componentes que estes substituem, podendo estes componentes possuírem ou não projetos idênticos. Uma vez que o sistema

[4]Em inglês: *recovery points.*

foi reparado, o serviço normal pode ser continuado.

2.3 Mecanismos de tratamento de exceções

Linguagens de programação modernas apresentam diversas características que podem ser utilizadas na construção de programas confiáveis. Dentre estas, podemos destacar a presença de um mecanismo de tratamento de exceções. Mecanismos de tratamento de exceções provêem uma estrutura adequada para a implementação de técnicas de tolerância a falhas em sistemas de software. Tratamento de exceções provê um adequado esquema para detectar e tratar erros e também incorporar atividades de tolerância a falhas em sistemas de software. A detecção de um erro resultará em uma exceção sendo levantada e um tratador de exceção, que implementa as atividade de recuperação de erros, sendo invocado.

A maior vantagem do uso de tratamento de exceções é que tal abordagem permite um alto grau de flexibilidade na recuperação de erros, pois ambos exceções e tratadores de exceções são livremente definidos pelo programador da aplicação e usados onde eles são necessários. Note que um tratador de exceção pode realizar uma restauração de estado (recuperação de erros por retrocesso) ou realizar apenas algumas operações no intuito de corrigir o estado do sistema (recuperação de erros por avanço). Conseqüentemente, o mecanismo de tratamento de exceções pode dar suporte a ambos os mecanismos de recuperação de erros.

Na literatura, há vários modelos para a implementação de mecanismos de tratamento de exceções em linguagens de programação. Nesta seção, apresentamos uma taxonomia para a classificação destes modelos [8, 9]. Os modelos serão classificadas segundo cinco aspectos: (1) representação da exceção, (2) localização do tratador, (3) associação entre exceções e tratadores, (4) continuação do fluxo de controle e (5) suporte para programação concorrente.

2.3.1 Representação das exceções

O primeiro aspecto a ser considerado é a representação das exceções, uma vez que o modelo de tratamento de exceções necessita de alguma representação interna que diferencia uma exceção em particular das demais.

Exceções representadas como cadeias de caracteres são adotadas em muitas linguagens, tais como CLU [79] e Guide [5, 70]. Exceções são implementadas como variáveis. Quando uma exceção é criada em uma operação O, o valor desta variável é alterado e o controle

retorna para a operação que invocou O, o qual tem a incumbência de testar o valor desta variável.

Exceções representadas como objetos de dados são adotadas, por exemplo, nas linguagens C++ [119] e Java [48]. Exceções são classes e instâncias destas classes são criadas sempre que uma exceção é levantada. Nesta abordagem exceções são levantadas por utilizar uma palavra chave da linguagem de programação.

Exceções representadas como objetos completos são adotadas, por exemplo, na linguagem Lore [38]. Exceções são organizadas em uma hierarquia de classes. Levantar uma exceção consiste em criar uma instância de uma classe e então chamar o método raise presente na interface deste objeto. Nesta abordagem, a exceção é um objeto que recebe mensagens posto que os específicos comportamentos de uma exceção são definidos como métodos na classe. Note que na segunda solução, os objetos são apenas utilizados como um repositório de dados, apesar da possibilidade de definição de métodos nas classes que definem estes objetos.

2.3.2 Associação entre exceções e tratadores

Após a exceção ser levantada, o fluxo de controle é interrompido e o controle é dado ao tratador associado à exceção. Esta associação é estática (definida em tempo de compilação) ou dinâmica se ocorreu durante a execução do programa.

A associação dinâmica é dependente do fluxo de execução do programa. Por exemplo, em PL/I [83] a associação é dinâmica. O comando ON especifica a associação, e este fica em efeito até um novo comando ON para esta mesma exceção ser executado ou o bloco de comandos em que ele se encontra ser finalizado. Para melhor ilustrar como é realizada esta associação é apresentado um exemplo na Figura 2.7(a). Neste exemplo, se a exceção OVERFLOW é levantada durante a execução de A, então Tratador 1 será executado. Alternativamente, se a mesma exceção é levantada durante a execução de B, o tratador associado a esta exceção será o Tratador 2.

Na associação estática, o código do tratador é associado com uma região de código do programa onde ocorre a execução normal. Desta forma, a separação entre a execução normal e a excepcional se torna mais visível. Um exemplo de um bloco associado estaticamente a tratadores encontra-se na Figura 2.7(b). Se alguma exceção é levantada durante a execução normal, o controle é passado para o tratador estaticamente associado a ela.

Outro tipo de associação ocorre através da cadeia dinâmica de chamadas de operações. Como mencionado anteriormente, um tratador pode ser associado estaticamente a uma região de código do programa. Entretanto, se esta região não tem um tratador para uma determinada exceção, então esta exceção deve ser propagada para uma região que dinamicamente a inclui. A Figura 2.8 ilustra um exemplo desse tipo de associação. Neste

```
ON OVERFLOW BEGIN
...          /* Tratador 1 */
END;
...          /* A */
ON OVERFLOW BEGIN
...          /* Tratador 2 */
END;
...          /* B */
```

(a) Associação dinâmica

```
BEGIN
... (Código execução normal)
... região protegida
EXCEPTION
... (Código execução anormal)
... tratadores de exceções
END
```

(b) Associação estática

Figura 2.7: Associação dinâmica e estática

exemplo, se uma exceção é levantada em *D*, o controle é passado para *Dex*. No entanto, se esta exceção não puder ser tratada por *Dex*, ela será propagada para *Cex* e assim sucessivamente. Neste mesmo exemplo, se uma exceção é levantada em *B*, o controle é passado imediatamente para *Aex* se esta possui um tratador para a exceção levantada.

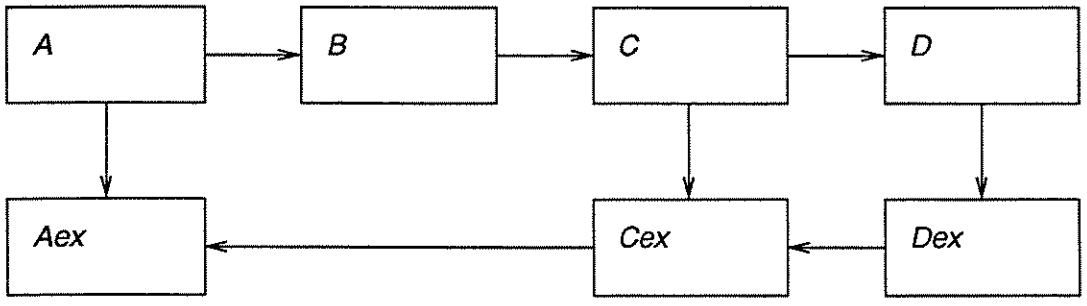


Figura 2.8: Associação através da cadeia dinâmica de chamadas de procedimento

2.3.3 Localização dos tratadores

Como descrito anteriormente, uma exceção pode estar associada estaticamente ou dinamicamente a um tratador. O local onde o tratador é associado com uma exceção é chamada de ponto de associação da exceção. Um ponto de associação pode ser: um comando, um bloco de comandos, um objeto ou uma classe.

A associação entre tratadores e um bloco de comandos ou entre tratadores e um comando é a solução mais encontrada nas diversas linguagens. Como exemplos, pode-se

citar a linguagem CLU [79] que associa os tratadores a comandos e C++ [119] e Java [48] que associa os tratadores a um bloco de comandos. Associar tratadores a classes permite definir um comportamento comum para todas as instâncias de uma classe em situações excepcionais. Como exemplos, as linguagens Guide [5, 70] e Lore [38] permitem este tipo de associação. E por último, é possível associar tratadores a objetos. Neste caso, dois objetos da mesma classe podem ter diferentes tratadores para uma mesma exceção; Cui e Gannon [32] descrevem uma implementação na linguagem de programação Ada [6] para este tipo de associação.

2.3.4 Continuação do fluxo de controle

Uma vez que uma exceção foi levantada, o fluxo de controle é interrompido e o controle é transferido ao tratador associado à exceção. Após a exceção ser tratada, o sistema pode retornar à sua execução normal. A questão levantada é: em que ponto o fluxo de controle é retornado ?

No modelo de terminação, a execução continuará no ponto seguinte ao qual a exceção foi tratada, e não seguinte ao ponto em que ela foi levantada. No modelo de continuação, o tratador tem a capacidade de retornar à atividade do componente ao ponto seguinte em que a exceção foi levantada.

Com propósito de ilustração é apresentado um exemplo (Figura 2.9) composto de três operações P, Q e R. P invoca Q que por sua vez invoca R. Durante a execução de R uma exceção é levantada e esta exceção é tratada por Q, supondo que não existe nenhum tratador local em R. Neste caso, a execução de R é suspensa e o controle é transferido ao tratador associado a esta exceção. Se o modelo de continuação é adotado, então R é retomado imediatamente após o ponto em que a exceção foi detectada. Caso contrário, isto é, se o modelo de terminação é adotado, a execução continuará logo após a exceção ser tratada. Neste exemplo, o controle é retornado à operação P.

2.3.5 Suporte para programação concorrente

Apesar de diversas linguagens concorrentes implementarem mecanismos de tratamento de exceções, apenas algumas destas permitem ativar tratadores nos diversos componentes quando uma exceção foi levantada em um deles. Exceções devem ser tratadas por mais de um componente de forma coordenada. Devido a própria natureza dos sistemas concorrentes, é possível que diversas exceções sejam levantadas ao mesmo tempo por diferentes componentes. Desta forma, é necessário um mecanismo de resolução de exceções que seleciona o tratador mais indicado a tratar todas as exceções levantadas. Nenhuma linguagem

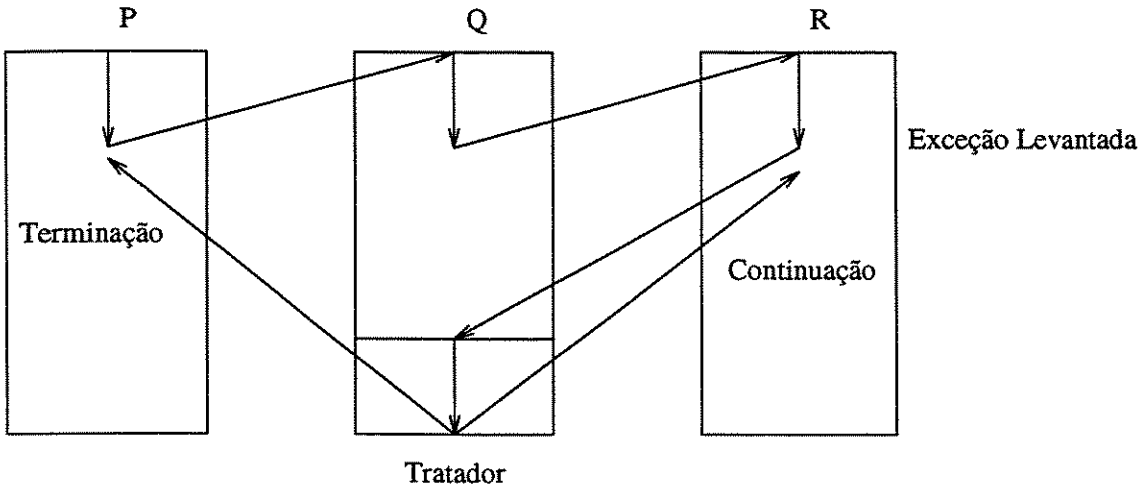


Figura 2.9: Semânticas do mecanismo de tratamento de exceções

de programação utiliza a árvore de exceção [27] discutida na Seção 2.5.2. A linguagem de programação Arche [16] permite resolução de exceções concorrentes através da definição de uma função de resolução que toma uma seqüência de exceções como parâmetro de entrada e retorna uma exceção que é denominada exceção ajustada. Após a função de resolução ter retornado a exceção ajustada, o tratador associado a esta exceção ajustada é ativada em todos os objetos que são implementações de um dado tipo. Desta forma, esta abordagem não é aplicável para a recuperação de erros coordenada entre múltiplos objetos de tipos diferentes.

2.4 Técnicas de tolerância a falhas em software

Nesta seção, nós consideraremos o problema de lidar com falhas em software. A principal causa de falhas de componentes de hardware são freqüentemente falhas físicas. Software, em contraste, não tem propriedades físicas; é uma entidade totalmente conceitual. Conseqüentemente, falhas físicas não podem existir em software. Falhas de software são sempre falhas de projeto. Isto é, uma falha de software é causada por projetos incorretos ou presença de *bugs*, que são usualmente causadas por erros humanos (em contraste para falhas físicas, que são causadas por leis físicas ou naturais). Para construir software que trata falhas de seus componentes, nós necessitamos de técnicas que lidam com falhas de projeto.

Como já discutido, um sistema consiste de vários componentes e alguns destes componentes podem ser falhos, i.e., podem existir falhas de projeto. O objetivo das técnicas de tolerância a falhas em software é assegurar que o sistema não falhe apesar da presença de

falhas de hardware ou software. Em outras palavras, a falha de algum dos componentes é mascarada e o defeito do sistema é tolerado. Todas estas técnicas são baseadas no uso de redundância. A replicação de componentes não é suficiente para lidar com falhas de projeto. Por essa razão, a diversidade de projeto é de fundamental importância e significa que os componentes são baseados em diferentes projetos. Existem duas técnicas básicas para organizar projetos diversos e construir software tolerante a falhas: blocos de recuperação e N-versões. Estas técnicas serão brevemente discutidas nas próximas seções.

2.4.1 Blocos de recuperação

A técnica de blocos de recuperação[5] foi proposta por Horning *et al.* [63] como uma forma para organizar diversos projetos e dar apoio à tolerância a falhas em software. Nossa discussão será baseada na descrição de blocos de recuperação apresentada por Randell [97].

Para cada componente crítico do sistema devem ser desenvolvidas versões alternativas, todas implementando as mesmas funcionalidades mas de forma distinta. Estas versões são combinadas com um teste de aceitação que verifica se o resultado da computação está de acordo com a especificação.

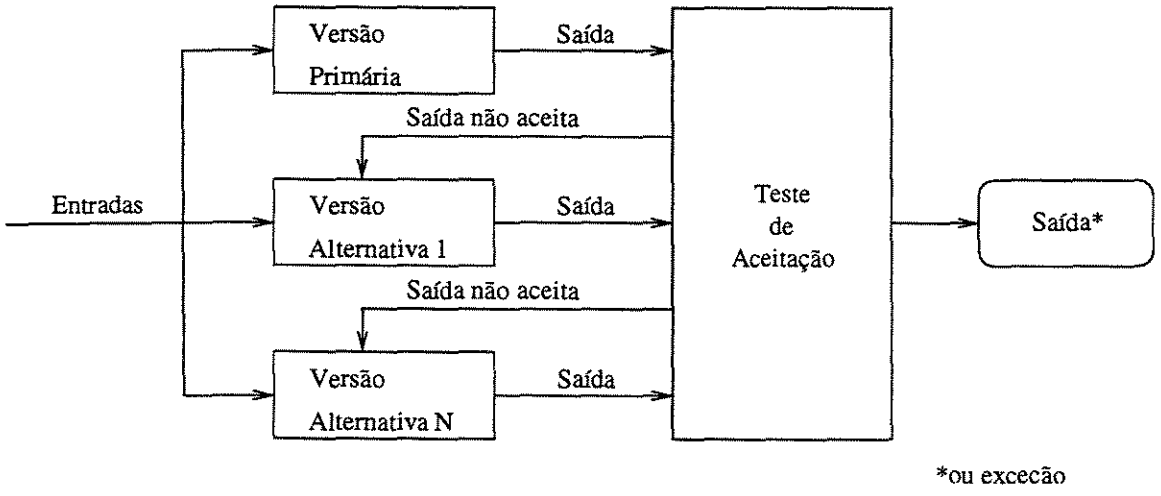


Figura 2.10: Bloco de recuperação

A execução inicia pela primeira versão (versão primária) e seu resultado é submetido ao teste de aceitação, se aceito, as demais versões não são executadas. Caso contrário a segunda versão (versão alternativa 1) é executada e submetida ao teste de aceitação. Para recuperação de erros, o bloco de recuperação emprega o mecanismo de recuperação de erros por retrocesso. Antes de executar a versão primária, um ponto de recuperação

[5]Em inglês: *recovery blocks*.

é estabelecido, salvando o estado do componente naquele ponto. Se um erro é detectado pelo teste de aceitação, o componente é restaurado ao estado salvo no momento em que o ponto de recuperação foi estabelecido. Caso isto aconteça, todos os efeitos desta versão são desfeitos e o sistema está em um estado consistente. Após a recuperação de erros ter sido realizada, uma nova alternativa é executada e o processo continua até que o resultado da execução de alguma versão seja aceito ou que tenham esgotadas todas as alternativas possíveis. Neste caso, uma exceção é levantada indicando que nenhuma versão passou pelo teste de aceitação (Figura 2.10).

Algoritmo 2.1 Definição de um bloco de recuperação

ensure <teste de aceitação>
by <módulo primário>
else by <módulo alternativo 1>
else by <módulo alternativo 2>
...
...
else error

Blocos de recuperação podem ser implementados com uma construção lingüística semelhante a encontrada no Algoritmo 2.1, onde a cláusula **ensure** especifica o teste de aceitação, a cláusula **by** especifica a versão primária, as cláusulas **else by** especificam as versões alternativas e a cláusula **else error** sinaliza uma exceção caso todas as alternativas falhem.

Algoritmo 2.2 Algoritmo de ordenação tolerante a falhas

ensure V[j+1] $\geq$ V[j] **for** j = 1,2,...,n-1
by sort V usando QuickSort
else by sort V usando InsertionSort
else by sort V usando BubbleSort
else error

Uma vez que a versão primária é a primeira a ser executada no bloco de recuperação, é normal utilizar como a primária, uma versão que tem características que fazem-na ser mais desejável em relação às outras versões. Por exemplo, a versão primária poderia ser a de menor tempo de execução ou a que consome menos memória. As versões alternativas deveriam realizar exatamente as mesmas funções que a versão primária, porém em uma diferente (talvez menos eficiente) implementação. O Algoritmo 2.2 ilustra bem estas características. Este bloco de recuperação foi projetado para ordenar um vetor V em ordem

crescente. A versão primária usa o que é esperado ser o algoritmo mais eficiente (em conseqüência, é o mais complexo) de ordenação. As versões alternativas também objetivam ordenar o vetor corretamente, porém usando algoritmos menos eficientes. Estes algoritmos serão mais simples do que o empregado pela versão primária e conseqüentemente estarão menos sujeitos a falhas de projeto.

2.4.2 Programação em N-versões

Programação em N-Versões[6] foi proposta por Avizienis [4] e consiste da utilização de N componentes de software, com $N \geq 2$, independentemente projetados a partir de uma especificação em comum. É uma extensão da técnica de RNM (Redundância N-Modular) discutida na Seção 2.1.4. A principal diferença entre estas duas técnicas é a utilização da diversidade de projeto no intuito de tolerar falhas de projeto. Cada componente é projetado independentemente. Isto deve ser feito preferencialmente por indivíduos ou grupos que não interagem entre si durante o processo de desenvolvimento e, sempre que possível, fazendo uso de algoritmos distintos.

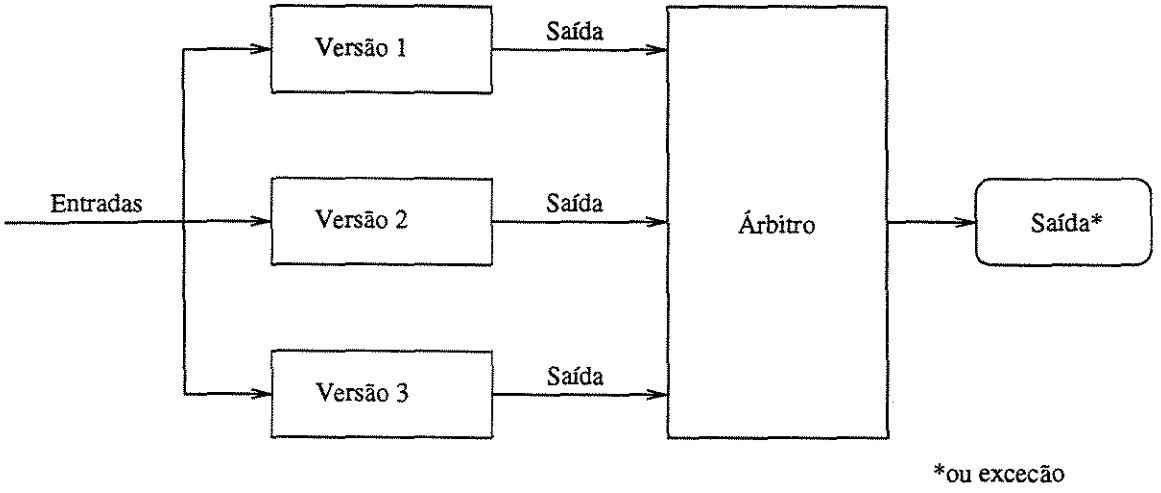


Figura 2.11: Programação em N-versões

As diferentes versões são executadas paralelamente e os resultados das versões são transmitidos ao árbitro, responsável por determinar um resultado consensual para a computação, de acordo com algum critério. Por exemplo, o árbitro pode fornecer como resultado consensual o resultado das versões que tiver ocorrido com maior freqüência. Isto é, o resultado será aquele dado pela maioria dos componentes. Caso o árbitro não seja

[6]Em inglês: *N-version programming.*

capaz de determinar o resultado, então uma exceção é sinalizada. A Figura 2.11 mostra um mecanismo de N-Versões composto por três versões de um componente e um árbitro.

2.5 Tolerância a falhas em sistemas concorrentes

Nesta seção, apresentaremos algumas técnicas de tolerância a falhas para sistemas concorrentes. Hoare [61] classifica um conjunto de processos concorrentes em três distintas categorias: independentes, competidores e cooperadores. A primeira categoria é um caso trivial; processos concorrentes são ditos serem independentes se o conjunto de recursos acessados por estes são disjuntos, desde que a atividade de cada processo deve ser completamente privada. Claramente, a provisão de tolerância a falhas para esta categoria é idêntica a provisão de tolerância a falhas em sistemas monoprocessados. Na segunda categoria, processos concorrentes são ditos serem competidores se competem no acesso a recursos comuns. E por fim, processos concorrentes são ditos serem cooperadores se eles estão cooperando para realizar uma determinada tarefa.

Descrevemos nesta seção o conceito de transação atômica que é usada para tolerar falhas em sistemas concorrentes competitivos, o conceito de conversações que são usadas para estruturar sistemas concorrentes cooperativos e incorporar tolerância a falhas de forma disciplinada e por fim, o conceito de ação atômica coordenada que consiste em uma tentativa de integrar transações atômicas e conversações em um mesmo arcabouço conceitual que lida com ambos tipos de concorrência (cooperação e competição).

2.5.1 Transação atômica

O conceito de transação atômica [58] originou-se de pesquisas em gerência de bancos de dados e, em conseqüência deste fato, inicialmente ficou restrito à área de banco de dados. Nos meados dos anos 80, aconteceram os primeiros experimentos que estendem o uso de transações atômicas para a programação de sistemas distribuídos; TABS [116] e Argus [78] estão entre os projetos pioneiros. Ambientes de programação distribuída transacionais e orientados a objetos, por exemplo, Arjuna [110] e Camelot/Avalon [42], têm reforçado a tese que motivou o uso de transações atômicas em ambientes distribuídos: ambientes transacionais oferecem uma abstração poderosa e eficiente para a programação de sistemas distribuídos tolerantes a falhas [25].

Em essência, uma transação atômica é uma operação tudo-ou-nada, a qual completa com sucesso tendo seu efeito pretendido ou falha inteiramente tendo nenhum efeito sobre o estado do sistema. Em termos mais formais, a transação atômica é dita ter as seguintes propriedades:

- Atomicidade de falhas: uma transação atômica completa com sucesso tendo seu efeito pretendido sobre o estado do sistema, ou falha inteiramente não tendo qualquer efeito sobre este. Esta propriedade garante que a computação termina em somente um de dois estados: a computação termina normalmente e produz o resultado desejado ou é abortada e não produz resultados. Mecanismos de recuperação de erros por retrocesso (Seção 2.2.3) são utilizados para garantir esta propriedade.

- Equivalência serial: o efeito da execução concorrente de duas ou mais transações atômicas é equivalente ao efeito de alguma execução serial. A implementação de um mecanismo de controle de concorrência é necessária para garantir esta propriedade.

- Permanência de efeito: os efeitos de transações atômicas são refletidos no estado do sistema e não perdidas como resultado de falhas subseqüentes. Esta propriedade garante que qualquer resultado produzido por uma transação atômica é armazenado em memória estável, um tipo de memória capaz de sobreviver, com alta probabilidade, a falhas de hardware. Um protocolo de validação da transação atômica[7] garante que todos os objetos modificados pela transação atômica têm seu estado escrito em memória estável, no caso da transação atômica ser confirmada, ou que as modificações não são gravadas, no caso de aborto.

Este modelo é muito efetivo para certas aplicações em que atividades concorrentes são independentes, necessitando competir no acesso a recursos comuns. Entretanto, uma das principais limitações do uso deste modelo é a falta de um suporte adequado para a cooperação entre atividades concorrentes [13, 131].

2.5.2 Conversação

O mecanismo de conversação [97] (Figura 2.12) consiste em uma extensão do bloco de recuperação (Seção 2.4.1) que pretende prover uma recuperação de erros coordenada para um conjunto de processos interativos e cooperativos. Algumas implementações podem ser encontradas na literatura [67, 68, 100, 109]. Cada processo que toma parte deve salvar seu estado ao entrar nela (isto é, estabelecer um ponto de recuperação). Enquanto participa da conversação, o processo pode apenas se comunicar com outros membros da conversação. Esta restrição na comunicação entre os processos limita a propagação de erros e elimina a possibilidade do efeito dominó [97]. Se algum processo falha em seu teste de aceitação, então uma exceção é levantada e cada processo deve tentar se recuperar da falha. O estado original de cada processo na conversação é restaurado e cada processo usa uma versão alternativa. Processos podem entrar em diferentes momentos, sendo necessário

[7] Em inglês: *commit protocol.*

apenas o estabelecimento de um ponto de recuperação para cada processo. No entanto, os processos têm que ser sincronizados (pelo menos logicamente) na saída implicando que o teste de aceitação em cada processo foi satisfeito [34].

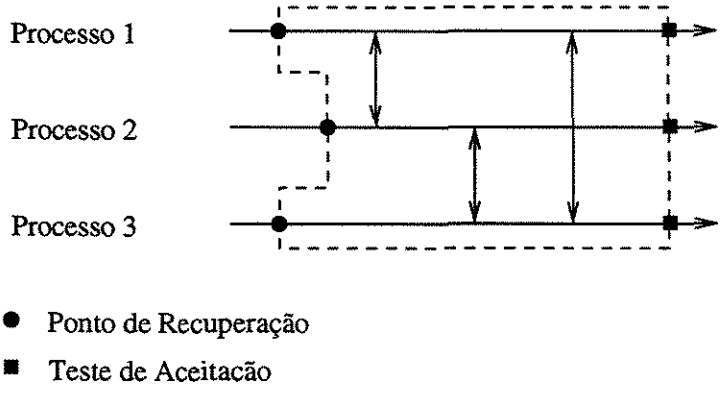


Figura 2.12: Descrição de uma conversação entre três processos

Muitas das técnicas de tolerância a falhas usam a propriedade de atomicidade. A conversação tem a restrição de que não existe nenhuma interação externa à conversação. Esta restrição garante a atomicidade da computação realizada pela conversação e portanto pode ser considerada uma ação atômica [74]. Conversações são principalmente usadas em sistemas orientados a processos (modelo de processos e mensagens [111]) e concentram-se principalmente em questões em como fatorar e organizar as interações entre processos sem endereçar o problema de consistência de recursos compartilhados acessados por estes processos.

Tratamento de exceções

Conversações podem usar ou recuperação de erros por avanço ou recuperação de erros por retrocesso ou uma combinação de ambos [27, 74]. Em qualquer caso, recuperação tem que ser coordenada e todos os participantes devem estar envolvidos. Se o mecanismo de recuperação de erros por retrocesso é utilizado, o estado original de cada processo na conversação é restaurado e cada processo ativa uma versão alternativa (diversidade de projeto). Uma abordagem para provisão de tratamento de exceções em conversações (ações atômicas) é descrito por Campbell e Randell [27]. Um conjunto de exceções é associado a cada conversação. Cada processo participando de uma conversação tem um conjunto de tratadores para (todas ou parte de) essas exceções. Quando uma destas exceções é levantada em algum processo, tratadores apropriados (para a mesma exceção em todos processos) serão iniciados em todos os participantes da conversação. Estes tratadores são projetados cooperativamente pelos programadores dos processos para recuperar o siste-

ma. Desta forma, os processos cooperam não apenas quando o sistema está operacional mas também quando o sistema precisa se recuperar de alguma falha.

Um mecanismo adicional de resolução de exceções é muito importante pois diversas exceções independentes podem ser levantadas ao mesmo tempo, ou diversos erros podem ser detectados os quais poderiam ser um sintoma de uma falha mais séria. A árvore de exceções[8] [27, 103] inclui todas as exceções associadas com a conversação e impõe uma ordem parcial nas exceções de tal modo que uma exceção em um nível mais alto está associada a um tratador que pode tratar qualquer exceção presente nos níveis inferiores desta hierarquia. Exceções podem ser de dois tipos na árvore de exceções: (i) exceções simples, ou (ii) exceções estruturadas. Exceções simples são as folhas da árvore de exceção. Exceções estruturadas são os nós não-folhas da árvore e correspondem a duas ou mais exceções sendo levantadas ao mesmo tempo.

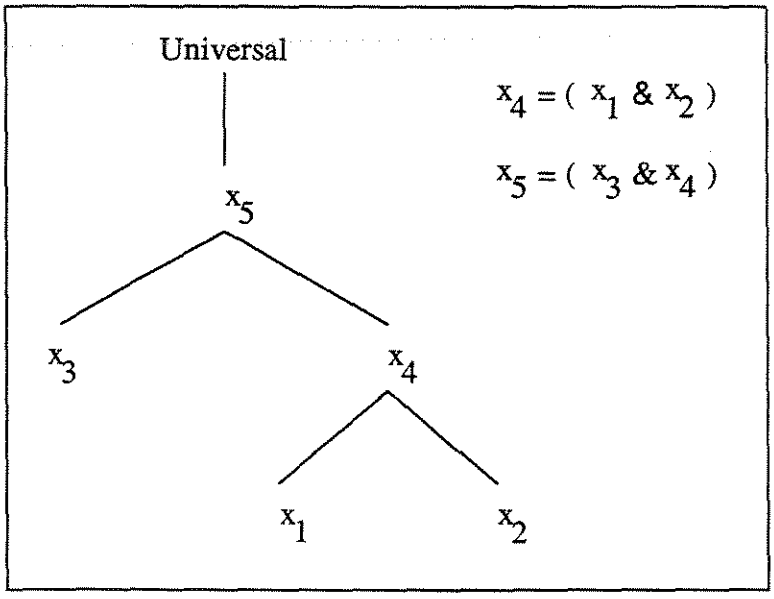


Figura 2.13: Exemplo de uma árvore de exceções

Por exemplo, considere a árvore de exceção ilustrada na Figura 2.13. Neste exemplo, apenas três exceções simples foram declaradas como possíveis a serem tratadas: x_1, x_2 e x_3. Cada exceção deste conjunto está associada a um tratador específico. Caso uma destas exceções seja levantada individualmente, o tratador associado a ela é ativado em todos os processos. No caso destas exceções serem levantadas concorrentemente, um tratador mais genérico deve ser ativado. Neste exemplo, se x_1 e x_2 são levantadas concorrentemente, então o tratador para a exceção estruturada x_4 será invocado. Analogamente, se as exceções x_1, x_2 e x_3 são levantados por diferentes processos, o tratador para a exceção

[8]Em inglês: *exception tree.*

estruturada x_5 é ativado em todos os processos no intuito de se recuperar deste erro. Caso uma exceção não declarada seja levantada, o tratador **Universal** é ativado. Este poderia ser associado a algum mecanismo de recuperação de erros por retrocesso, uma vez que esta exceção não foi prevista.

2.5.3 Ação atômica coordenada

O conceito de ação atômica coordenada [98, 130] é a primeira tentativa de integrar transações atômicas e conversações em um mesmo arcabouço conceitual que lida com ambos tipos de concorrência (cooperação e competição). Uma ação atômica coordenada consiste de uma atividade de um grupo de objetos que combina algumas propriedades de conversações, transações atômicas e tratamento de exceções no modelo de objetos. Conversações são usadas para controlar concorrência cooperativa e implementar coordenada e disciplinada recuperação de erros, enquanto que transações atômicas são usados para manter a consistência de recursos compartilhados na presença de falhas e concorrência competitiva.

Cada ação atômica coordenada consiste de um conjunto de papéis que são ativados concorrentemente por participantes. Os participantes cooperam no escopo de uma ação atômica coordenada por realizar alguma computação em um conjunto de objetos externos. Uma ação atômica coordenada inicia a sua computação quando todos papéis são ativados e termina quando cada papel termina a sua execução. Objetos que são externos à ação atômica que podem conseqüentemente ser acessados concorrentemente por diversas ações atômicas coordenadas devem apresentar semânticas transacionais. Isto é, a seqüência de operações realizadas por uma dada ação atômica coordenada em um conjunto de objetos externos deve ser atômica em relação às outras ações atômicas coordenadas. Em adição, ações atômicas coordenadas podem usar objetos locais da ação atômica coordenada que são utilizados pelos participantes para trocar informações entre si. Como podem ser acessadas por diversos participantes, a consistência destes objetos deve ser assegurada. Isto é realizado pelos próprios objetos por implementar as semânticas de um monitor.

Ações atômicas coordenadas provêem um arcabouço básico para tratamento de exceções que dá apoio a uma variedade de mecanismos de tolerância a falhas objetivando tolerar tanto falhas de hardware quanto de software. Falhas de hardware são toleradas por usar transações atômicas que assegura que os efeitos da ação atômica coordenada em objetos externos são permanentes. Falhas de software são toleradas por usar diversidade de projeto. Durante a execução de uma ação atômica coordenada, um dos participantes pode levantar uma exceção. Se aquela exceção não pode ser tratada localmente, então esta deve ser propagada para os outros participantes da ação atômica coordenada. Dado que é possível que diferentes exceções sejam levantadas ao mesmo tempo, um processo de resolução de exceções [27, 103] é necessário. Uma vez que a exceção resolvida foi pro-

pagada para todos os participantes, então alguma forma de recuperação de erros deve ser invocada. Pode ainda ser possível completar a execução da ação atômica coordenada usando recuperação de erros por avanço. Ou alternativamente, usar recuperação de erros por retrocesso e desfazer os efeitos da ação atômica coordenada e iniciar novamente a sua execução possivelmente utilizando um diferente projeto para cada papel.

O conceito de ações atômicas coordenadas é recursivo. Isto é, ações atômicas coordenadas aninhadas podem ser usadas para estruturar a execução de uma ação atômica coordenada. Os efeitos de uma ação atômica coordenada aninhada apenas torna-se permanente quando a ação atômica coordenada que a contém termina. Entretanto, os efeitos de uma ação atômica coordenada aninhada após terminada são visíveis no escopo da ação atômica coordenada que a contém.

Sincronização dos participantes

Participantes podem entrar em um ação atômica coordenada de forma assíncrona mas eles têm que ser sincronizados (pelo menos logicamente) na saída [34]. Sincronização na saída envolve em concordar no sucesso ou falha da ação atômica coordenada e então confirmar os efeitos nos objetos externos ou tentar recuperar dos erros que são detectados pelo teste de aceitação.

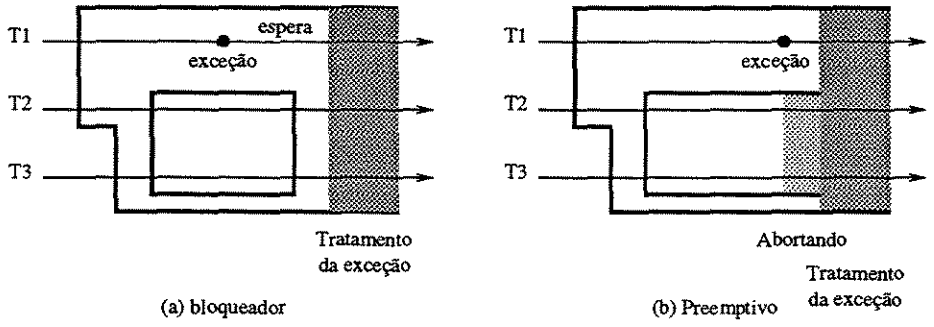


Figura 2.14: Bloqueador X preemptivo

Com respeito ao tratamento de exceções existem dois distintos esquemas: bloqueadores e preemptivos (Figura 2.14). Em esquemas bloqueadores, cada participante ou finaliza suas atividades com sucesso ou falha levantando uma exceção. Entretanto, os outros participantes envolvidos nesta ação atômica coordenada são informados da exceção apenas quando eles finalizam suas atividades ou quando também detectam uma exceção. Em contraste, esquemas preemptivos usam algumas características presentes em linguagens de programação para interromper todos os participantes quando uma exceção é detectada.

No esquema bloqueador, a provisão da recuperação de erros e resolução de exceções concorrentes é facilitada desde que cada participante está em um estado consistente quan-

do os tratadores são invocados. Além disso, não existe a necessidade de abortar ações atômicas coordenadas aninhadas desde que as ações aninhadas já foram finalizadas com sucesso ou tiveram seus erros tratados pelos tratadores associados às ações atômicas coordenadas aninhadas.

Em esquemas preemptivos, é obrigatória a presença de algum mecanismo de interrupção assíncrona dos participantes que geralmente não está presente em linguagens de programação. Além disso, é mais difícil analisar, entender e testar programas que usam tal mecanismo [102]. Outra limitação com esquemas preemptivos é a dificuldade em abortar ações atômicas coordenadas aninhadas. Em particular, mesmo que programadores implementem o aborto das ações atômicas coordenadas aninhadas, um protocolo sofisticado (como exemplo, o protocolo definido por Romanovsky *et al.* [103]) deve ser utilizado para levantar exceções de aborto em todas as ações aninhadas (recursivamente e seguindo a ordem correta).

2.6 Considerações finais

Neste capítulo apresentamos os fundamentos de tolerância a falhas. Na Seção 2.5 consideramos o problema de lidar com falhas em sistemas concorrentes. Primeiramente, apresentamos o conceito de transação atômica que é efetivo para tolerar falhas em sistemas concorrentes competitivos mas que falta um suporte adequado na definição de atividades concorrentes cooperativas. Então, apresentamos o conceito de conversação que provê uma infra-estrutura adequada para o desenvolvimento de aplicações concorrentes cooperativas. No entanto, este modelo não se preocupa com questões em como manter a consistência de recursos compartilhados acessados pelas atividades concorrentes. E por fim, apresentamos o conceito de ação atômica coordenada que integra transações atômicas e conversações em um mesmo arcabouço conceitual que lida com ambos tipos de concorrência (cooperação e competição).

Diversas aplicações foram modeladas usando ações atômicas coordenadas desde a sua proposta [130]; em particular, uma série de células de produção [137], incluindo uma em que falhas de diversos dispositivos teriam que ser toleradas [132] e uma com restrições de tempo real [104], e um sistema de leilões eletrônicos [127].

Recentemente, Vachon *et al.* [127] propôs uma extensão ao conceito de ação atômica coordenada através da introdução de papéis assíncronos. Esta extensão permite que diferentes participantes ativem o mesmo papel de forma concorrente. O exemplo de papel assíncrono descrito neste trabalho é o papel de comprador[9] em um sistema de leilões eletrônicos. Tal papel é ativado pelos inúmeros participantes que desejam comprar o

[9]Em inglês: *bidder*.

item posto em leilão. Segundo a abordagem descrita neste artigo, um leilão consiste de uma ação atômica coordenada composta por um participante desempenhando o papel de vendedor de algum item e um número de participantes que desempenham o papel assíncrono de comprador. Em nossa opinião, o conceito de ação atômica coordenada não é o mais adequado para a modelagem deste tipo de aplicação desde que os participantes que desempenham o papel assíncrono comprador não cooperam e sim competem no acesso ao vendedor (submissão de lances). Acreditamos que a utilização do modelo cliente-servidor facilitaria a modelagem deste tipo de aplicação ao mesmo tempo que proporcionaria os mesmos resultados. Concluindo esta discussão, gostaríamos de salientar que acreditamos que a técnica de ações atômicas coordenadas é bastante poderosa na modelagem de um grande leque de aplicações. No entanto, temos que ter consciência que existem algumas aplicações em que a utilização deste conceito não é a mais adequada. Por outro lado, existem diversos fatores que complicam a utilização desta técnica (bem como as outras técnicas discutidas neste capítulo), tais como: (i) linguagens de programação não dão suporte ao emprego delas diretamente; (ii) tal técnica é geralmente empregada na fase de implementação do sistema; e (iii) projetistas de aplicações necessitam levar em conta muitos detalhes de implementação, e portanto, dificulta o seu entendimento.

Acreditamos que se as técnicas de tolerância a falhas discutidas neste capítulo fossem documentadas de tal forma que facilitassem o seu entendimento e se existisse uma abordagem sistemática na incorporação de características de tolerância a falhas durante o ciclo de desenvolvimento do sistema, a construção de sistemas tolerantes a falhas seria bastante facilitada.

Neste contexto, Capítulo 4 apresenta uma arquitetura de software genérica para introduzir atomicidade, redundância de software, tratamento de exceções e recuperação de erros coordenada no desenvolvimento de sistemas confiáveis e Capítulo 6 apresenta uma abordagem sistemática na incorporação de características de tolerância a falhas durante o ciclo de desenvolvimento do sistema, iniciando-se na fase de projeto arquitetural passando pelo projeto detalhado e terminando na fase de codificação do sistema.

Capítulo 3

Técnicas de Engenharia de Software para Estruturação de Software

Um dos principais objetivos deste trabalho é a utilização sistemática de recentes técnicas de engenharia de software para a estruturação de software com associadas características de tolerância a falhas. Este capítulo introduz brevemente algumas técnicas de engenharia de software para estruturação de software empregadas nesse trabalho com intuito de facilitar a compreensão dos capítulos subseqüentes.

3.1 Processo de desenvolvimento de software

O processo de desenvolvimento de software pode ser definido como o conjunto de todas as atividades relacionadas ao desenvolvimento, controle, validação e manutenção de um software operacional. Dessa forma, a disciplina de engenharia de software tem como objetivo consolidar o entendimento deste processo, desenvolver modelos e formalismos adequados para representá-lo, além de produzir estratégias, metodologias e ferramentas para suportar a execução e manutenção deste processo. A utilização de um processo bem definido permite um efetivo monitoramento e controle do desenvolvimento do software, podendo levar à redução dos custos de produção e à melhoria da qualidade e integridade do software [94].

O conjunto canônico de fases (Figura 3.1) do ciclo de vida de um software é considerado como os seguintes:

- Elaboração dos requisitos. Uma vez que o software sempre faz parte de um sistema mais amplo, o trabalho inicia-se com o estabelecimento dos requisitos para todos os elementos do sistema e prossegue com a atribuição de certo subconjunto desses requisitos ao software.

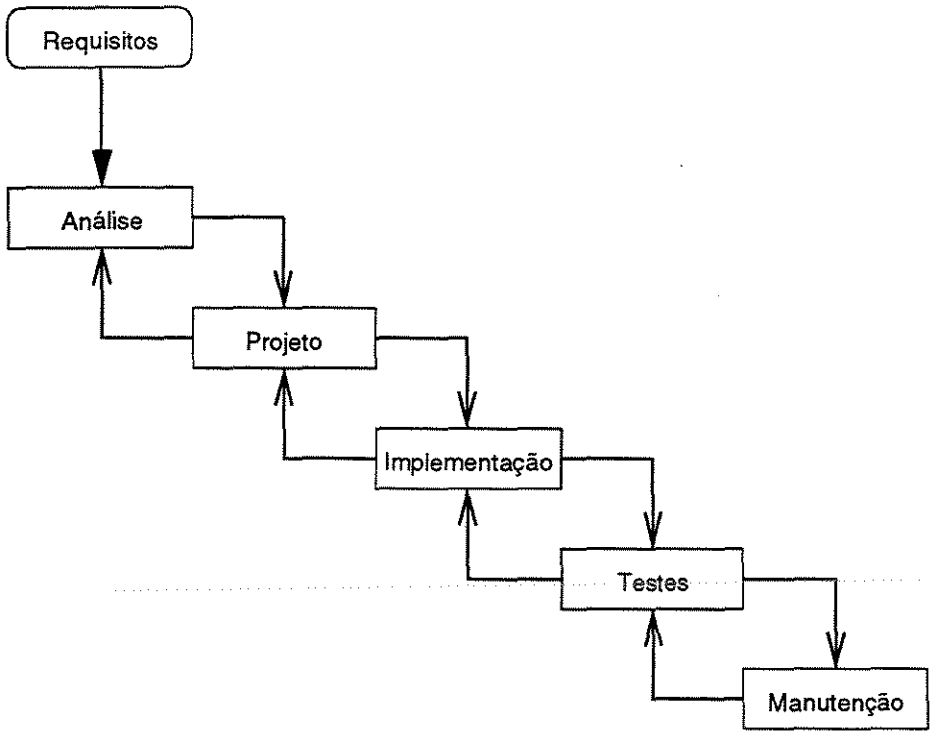


Figura 3.1: Fases do ciclo de vida de um software

- Análise de requisitos. É o primeiro passo técnico do processo de desenvolvimento de software. É nesse ponto que uma declaração geral do escopo do software é aprimorada numa especificação concreta que se torna base para todas as fases do processo de desenvolvimento de software que se seguirão.

- Projeto. O projeto de software é o processo pelo qual os requisitos são traduzidos numa representação do software. Inicialmente, a representação descreve uma visão global do software. Subseqüentes refinamentos levam a uma representação que está próxima ao código fonte. A fase de projeto envolve o projeto de alto-nível, especificando a arquitetura do sistema, e o projeto detalhado (refinamento) de cada elemento pertencente à arquitetura.

- Implementação. A fase de implementação/codificação do processo de desenvolvimento de software é um processo de tradução em que o projeto detalhado do software é convertido numa linguagem de programação que, por fim, é transformada em instruções executáveis por máquina.

- Testes. Assim que o código for gerado, iniciar-se-á a realização de testes do software. O processo de realização de testes concentra-se nos aspectos lógicos internos do

software, garantindo que todas as instruções tenham sido testadas, e concentra-se também nos aspectos funcionais externos, ou seja, realizando testes para descobrir erros e garantir que a entrada definida produza resultados reais que concordem com os resultados exigidos.

- Manutenção. Indubitavelmente,o software sofrerá mudanças depois que entre em operação. Ocorrerão mudanças porque erros foram encontrados, porque o software deve ser adaptado a fim de acomodar mudanças em seu ambiente externo ou porque o cliente exige acréscimos funcionais ou de desempenho. A manutenção de software reaplica cada uma das fases precedentes a um software existente, e não a um novo.

O processo de desenvolvimento de software geralmente beneficia-se da utilização de metodologias baseadas em orientação a objetos, que envolvem desde a fase de análise de requisitos até a fase de manutenção do software . A abordagem orientada a objetos para a construção de sistemas permite que um mesmo conjunto de conceitos e notações seja utilizado através de todo o ciclo de vida do software.

Dessa forma, metodologias orientadas a objetos oferecem um melhor acompanhamento do processo de desenvolvimento de software e meios para a reutilização de software permitindo a construção de sistemas mais confiáveis, devido a características inerentes ao próprio modelo de objetos, tais como, abstração de dados, encapsulamento, herança e polimorfismo.

3.1.1 O modelo de objetos

O modelo de objetos representa em software, através das noções de classes, objetos e herança, entidades que encontramos no mundo real. Uma classe descreve um conjunto de objetos com a mesma estrutura e o mesmo comportamento. O comportamento de um objeto é caracterizado pela sua interface, ou seja, o conjunto de operações e atributos públicos da sua classe. Um objeto é composto de atributos e de operações que são realizadas sobre estes atributos. Ao receber uma mensagem, o objeto executa a operação solicitada, através de computações sobre seus próprios atributos e, possivelmente, requisitando a execução de operações de outros objetos. Herança é um mecanismo para derivar novas classes a partir das classes já existentes através de um processo de refinamento. Uma classe derivada herda a representação dos atributos e operações públicas da classe base, mas pode adicionar novas operações, estender a representação dos atributos ou sobrepor a implementação de operações já herdadas. Deste modo, o modelo de objetos baseia-se na definição de classes e na criação de hierarquias de classes, onde as propriedades comuns podem ser transmitidas das superclasses para as subclasses através do mecanismo de herança.

O crescente interesse por esse modelo se deve ao fato dele proporcionar uma abordagem promissora para a estruturação e construção de sistemas complexos, dado que ele facilita a reutilização, modificação e extensão do software, melhorando a qualidade e reduzindo o custo de desenvolvimento do software.

UML

Várias metodologias orientadas a objetos podem ser encontradas na literatura [19, 65, 106]. Tais metodologias apresentam diferentes notações e regras muitas vezes conflitantes. De fato, não existia uma representação padrão para os conceitos de orientação a objetos nessas diferentes metodologias. Entretanto, esforços de padronização levaram à proposta de uma nova linguagem para modelagem de sistemas - UML[1] [20].

UML representa a unificação das metodologias de Booch [19], OMT [106] e OOSE [65]. Esta unificação teve como principais objetivos: (i) eliminar elementos que não eram utilizados na prática; (ii) adicionar elementos eficazes de outras metodologias; e (iii) criar novos elementos para as soluções não disponíveis. Diante destas perspectivas, adotamos neste trabalho UML como a linguagem de modelagem de sistemas.

UML é uma linguagem para especificação, construção, visualização e documentação dos artefatos de um sistema de computação [20]. Artefatos englobam os conceitos e modelos que apóiam o desenvolvimento de uma estrutura; bem como os métodos, que descrevem como trabalhar com estes conceitos e modelos para um desenvolvimento ideal [65]. UML pretende ser uma linguagem universal para modelagem de sistemas, isto é, pretende expressar modelos de vários tipos de aplicações com propósitos variados. A existência de um processo universal simples para todos os estilos de desenvolvimento não parece possível: o que funciona para um projeto compacto de software provavelmente não serve para um tipo de sistema de grande porte, distribuído e crítico. Entretanto, UML pode ser usada para expressar os artefatos de todos estes diferentes processos através de modelos que são produzidos.

Entretanto, para obter melhores resultados, o projetista deveria considerar um processo que é:

- **orientado a casos de uso** que significa que casos de usos são utilizados como o primeiro artefato para estabelecer o comportamento desejado do sistema e para verificar e validar a arquitetura do sistema.
- **centrado numa determinada arquitetura** que significa que a arquitetura do sistema é usada como o primeiro artefato para conceituar, construir, gerenciar, e evoluir o sistema em desenvolvimento.

[1] *Unified Modeling Language.*

- **iterativo e incremental** onde o sistema é desenvolvido através de sucessivos refinamentos. A essência do processo iterativo é que a especificação do sistema evolui à medida que o software vai sendo desenvolvido. Um bom exemplo de um processo iterativo e incremental é o modelo em espiral [18, 94, 115]. O processo se move sob uma espiral evolucionária onde as fases do do ciclo de desenvolvimento do software são executadas de forma iterativa e incremental.

Os detalhes do processo geral de desenvolvimento devem ser adaptados para a cultura particular de desenvolvimento ou para o domínio da aplicação de uma organização específica. Esta separação entre a linguagem de modelagem e os processos possibilita aos usuários da metodologia um grau considerável de liberdade para desenvolver um processo específico usando uma linguagem comum de expressão.

Os principais recursos que os desenvolvedores utilizam para manipular modelos são os diagramas. Um diagrama é a projeção dos elementos de um modelo; os mesmos elementos podem aparecer em múltiplos diagramas. Um diagrama pode apresentar alguns - mas não necessariamente todos - detalhes de um elemento particular. UML define um conjunto central de diagramas, os quais provêem múltiplas perspectivas de um sistema que se encontra sob análise ou desenvolvimento. Uma descrição detalhada desse conjunto de diagramas encontra-se em Booch *et al.* [20]. Não temos a intenção de descrevê-los neste trabalho, apenas faremos breves comentários:

- Diagrama de casos de uso, que é utilizado para delimitar o sistema e definir a funcionalidade que ele poder oferecer;
- Diagramas de estrutura estática:
  - Diagrama de classes, que mostra a estrutura estática do sistema através da descrição de seus componentes (classes, objetos, módulos, composições, etc) e seus relacionamentos.
  - Diagrama de objetos, que representa uma instância de um diagrama de classes. Ele mostra um quadro do estado de um sistema em um determinado momento.
- Diagramas de comportamento:
  - Diagrama de estados, que descreve a evolução temporal (seqüência de estados) de um objeto de uma determinada classe, em resposta às interações com outros objetos internos ou externos ao sistema.
  - Diagrama de seqüências, que é utilizado para modelar cenários, ilustrando as principais interações entre objetos.

  - Diagrama de colaborações, que mostra a seqüência de mensagens que implementa uma operação ou uma transação envolvendo um conjunto de objetos.

- Diagramas de implementação:

  - Diagrama de componentes, que descreve o projeto físico de um sistema, ou seja, os componentes de software e hardware que implementam o projeto lógico.
  - Diagrama de disposições, que especifica a topologia física na qual um sistema executa.

3.2 Arquiteturas de software

O modelo de objetos apresentado na seção 3.1.1 é um modelo promissor para um desenvolvimento de software pois oferece meios para a reutilização de software e que permite a construção de sistemas mais confiáveis, devido a características inerentes ao próprio modelo de objetos, tais como, abstração de dados, encapsulamento, herança e polimorfismo. No entanto, atualmente, somente o emprego da tecnologia de orientação a objetos não é mais suficiente para lidar com o desenvolvimento de sistemas de software com graus de complexidade elevados. Neste contexto, somente a aplicação de metodologias orientadas a objetos não proporciona um nível de reutilização e de facilidade de evolução apropriado [72]. Dessa forma, no intuito de controlar a complexidade dos sistemas, torna-se necessário, primeiramente realizar um projeto que descreva a organização do sistema, também chamada de arquitetura de software. Uma arquitetura de software engloba a definição de estruturas gerais de um projeto de software, descrevendo os elementos que compõem o sistema e as interações entre estes.

Arquiteturas de software fornecem uma descrição abstrata de um sistema por focar em sua estrutura e abstrair dos detalhes de implementação. As propriedades não-funcionais de um sistema são largamente restringidas ou permitidas pela arquitetura de software deste sistema. A arquitetura de software de um sistema é decidida durante as primeiras fases do desenvolvimento do sistema, em que uma abordagem para solucionar um específico problema é selecionada.

Se uma arquitetura de software apropriada é escolhida (em particular que dá suporte a tolerância a falhas) durante a fase de projeto arquitetural, então um uso consistente de técnicas de tolerância a falhas através todo o ciclo de desenvolvimento do sistema é obtido.

3.2.1 Componentes, conectores e estilos arquiteturais

A descrição de uma arquitetura de software é baseada nos seguintes conceitos [108]:

- Componentes lógicos que representam as unidades de computação e que tem pontos de interface (portas de componentes[2]) com outros elementos arquiteturais;
- Conectores que representam o padrão de composição entre os componentes lógicos e que tem pontos de interface (papéis de conectores[3]) com outros elementos arquiteturais. Um conector prescreve o protocolo de interação que ocorre entre os componentes que são compostos através dele; e
- Configuração que define a estrutura do sistema por compor uma coleção de instâncias de componentes conectados através de instâncias de conectores. Uma arquitetura de software é então definida como uma configuração que instancia componentes e conectores. Isto é, uma arquitetura de software define o sistema em função de seus componentes lógicos e da interação entre esses componentes.

Muitos dos métodos tradicionais de desenvolvimento de software não incorporam um aspecto muito importante, que é a reutilização sistemática de conhecimento prévio sobre sistemas relacionados na definição da arquitetura de software de um sistema. A noção de estilos arquiteturais provê meios de explorar as semelhanças entre sistemas. Um estilo arquitetural define um conjunto de propriedades compartilhados pelas arquiteturas de software que são membros deste estilo arquitetural. Um estilo arquitetural caracteriza uma família de sistemas que são relacionadas pelo compartilhamento de propriedades estruturais e comportamentais (semântica). O interesse por um determinado estilo está relacionado com a sua habilidade de encapsular tipos importantes de decisões de projeto e de enfatizar importantes requisitos sobre os componentes lógicos. Um estilo de arquitetura define um vocabulário de componentes lógicos e tipos de conectores, mais um conjunto de restrições sobre a combinação destes.

Entre os exemplos de estilos arquiteturais encontrados na literatura podemos citar estes:

- **Cliente & servidor.** Neste estilo, existem dois tipos de componentes: **clientes** e **servidores. Chamadas de procedimentos remotos** podem ser usados para prover comunicação entre estes componentes. Um sistema organizado de acordo com este estilo é constituído de um conjunto de **clientes** que requisitam serviços (usando chamadas de procedimentos remotos ou outra forma de comunicação) a **servidores.**

[2]Em inglês: *component ports.*
[3]Em inglês: *connector roles.*

- **Organização orientada a objetos.** Os componentes deste estilo são objetos (instâncias de tipos abstratos de dados). Objetos são responsáveis por preservar a integridade de sua representação e esta representação é escondida de outros objetos. Objetos interagem através de invocações de métodos.

- **Camada.** Este estilo arquitetural [108] define uma estrutura hierárquica em que cada camada provê serviços para a camada superior e usa os serviços da camada inferior na hierarquia. Protocolos em rede [123] são provavelmente os melhores exemplos de arquiteturas em camadas.

Existem diferentes maneiras de se descrever, representar e especificar uma arquitetura de software. Podem ser utilizadas descrições informais ou formais. As descrições informais são tradicionalmente feitas através de gráficos, com caixas representando os componentes arquiteturais e linhas representando os conectores. Normalmente essas descrições são auxiliadas por explicações feitas em linguagem natural. Também existem alguns tipos de descrições que utilizam linguagens de especificações mais formais e rigorosas, chamadas de linguagens de definição de arquiteturas[4]. Entre os exemplos de linguagens de definição de arquiteturas encontrados na literatura podemos citar estes: Aesop [55], Rapide [80], Unicon [107] e Wright [2].

3.2.2 Propriedades não-funcionais X propriedades funcionais

Uma propriedade (ou requisito) funcional lida com algum aspecto particular da funcionalidade do sistema, e está usualmente relacionada a um requisito funcional especificado. No passado, os desenvolvedores de software concentravam-se principalmente em proporcionar as propriedades funcionais do sistema. Entretanto, nos sistemas de software modernos, as propriedades não-funcionais estão se tornando cada vez mais importantes. Uma propriedade não-funcional denota uma característica de um sistema que não é coberta pela sua especificação. Uma propriedade não-funcional tipicamente está relacionada com aspectos relacionados com, por exemplo, confiabilidade, adaptabilidade, interoperabilidade, persistência e segurança. As propriedades não-funcionais de uma arquitetura de software têm um grande impacto no seu desenvolvimento, manutenção e extensão. Quanto maior e mais complexo um sistema de software for e quanto maior for seu tempo de vida, maior será a importância das suas propriedades não-funcionais.

Como conseqüência, as decisões de projeto que devem ser tomadas no desenvolvimento de uma arquitetura de software geralmente são complexas e envolvem vários aspectos relacionados com as suas propriedades não-funcionais. Com o objetivo de efetuarmos boas

[4]Em inglês: *Architecture Definition Languages (ADLs).*

decisões de projeto, é essencial que possamos esclarecer o contexto do problema para que possamos identificar as melhores escolhas dentre as diversas opções de projeto possíveis. Um modo de clarificarmos este contexto é aplicarmos a técnica de separação de interesses[5]. Para que as diversas tarefas envolvidas no projeto de uma arquitetura de software possam ser separadas, é necessário inicialmente o estabelecimento dos limites conceituais entre elas. Cada partição obtida será responsável por resolver um subconjunto das propriedades propostas, e o projetista da arquitetura é então responsável pela composição das diferentes partições de modo que todas as propriedades sejam satisfeitas em conjunto.

3.2.3 Arquiteturas de software no desenvolvimento de sistemas

A arquitetura de software serve de base para a fase seguinte que envolve o projeto detalhado dos componentes lógicos e dos conectores. Após o projeto detalhado inicia-se a fase de implementação/codificação. Nesta fase, pode-se utilizar componentes de software (Seção 3.5) existentes que realizam as funcionalidades descritas no projeto, ou implementar novos componentes de software para essa aplicação. Após a fase de implementação, realiza-se a fase de integração das implementações de cada componente lógico, de acordo com a arquitetura previamente definida. Nesta fase, é verificada a compatibilidade dos componentes implementados com a arquitetura, além disso, deve-se assegurar que a arquitetura implementada está de acordo com os requisitos iniciais e provê as propriedades esperadas.

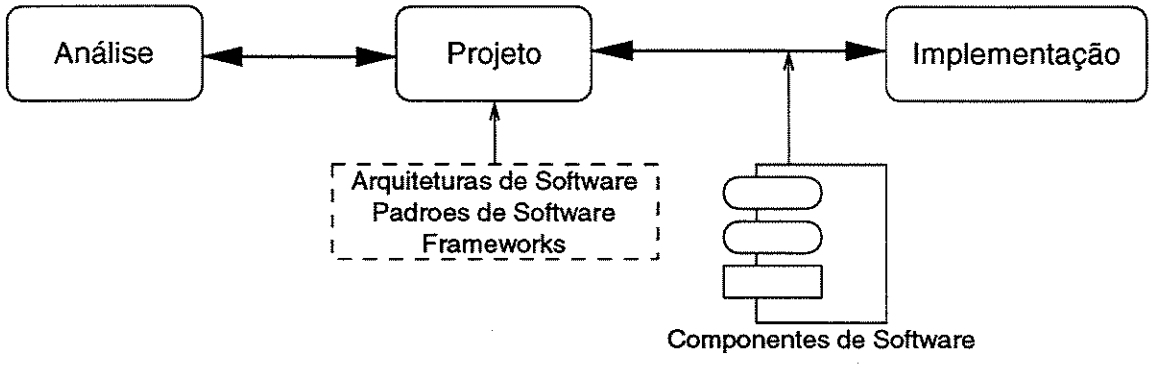


Figura 3.2: Técnicas de engenharia de software no processo de desenvolvimento de software

A construção da arquitetura do sistema inicia-se na fase de projeto do ciclo de vida do desenvolvimento de software. Seu propósito é definir a estrutura geral do sistema e prover as primeiras soluções para as propriedades funcionais e não-funcionais, como reusabilidade, portabilidade e confiabilidade. Se estas soluções não forem definidas nas decisões

[5]Em inglês: *separation of concerns.*

arquiteturais, dificilmente essas propriedades serão alcançadas durante a implementação do sistema. Portanto, as decisões quanto à arquitetura do sistema são difíceis de serem tomadas e, na maioria das vezes, requerem grande experiência por parte do arquiteto do sistema, sobre engenharia de software e sobre o domínio do sistema em consideração. Uma maneira de reutilizar essa experiência é através da sua documentação na forma de padrões de software.

3.3 Padrões de software

Como discutido anteriormente, a noção de estilos arquiteturais provê meios de reutilizar sistematicamente o conhecimento prévio sobre sistemas relacionados na definição da arquitetura de software de um sistema. Por outro lado, os padrões de software permitem que soluções genéricas de software previamente utilizadas e testadas sejam reutilizadas, levando ao desenvolvimento de aplicações elegantes e reutilizáveis. Eles capturam a experiência dos projetistas, permitindo que a mesma seja repassada a outros, aumentando o seu grau de reutilização. Os padrões atuam como uma solução para um problema que ocorre repetidamente em um determinado contexto.

Os padrões de software foram inicialmente introduzidos no campo de orientação a objetos [49], mas atualmente são utilizados em muitos outros domínios. Buschmann [24] *et al.* subdivide os padrões de software em três categorias que representam diferentes níveis de abstração:

- Padrões arquiteturais. Relacionam-se com a organização geral do sistema. Esses padrões definem um conjunto pré-definido de componentes lógicos de alto nível, especificando seus relacionamentos e estabelecendo regras e diretrizes para a organização dos relacionamentos entre eles. Os padrões arquiteturais podem ser utilizados para descrever os estilos arquiteturais descritos na Seção 3.2. Um exemplo de padrão arquitetural é o padrão Reflexão [24]. Este padrão define uma arquitetura reflexiva, que divide uma aplicação em basicamente dois níveis, o meta-nível e o nível base (Seção 4.3). No meta-nível residem os componentes do sistema responsáveis pelas ações administrativas a serem realizadas sobre um sistema alvo localizado no nível base, onde residem os componentes que lidam com a funcionalidade básica da aplicação. A ligação entre os níveis é feita através de um mecanismo de interceptação e materialização de mensagens, de tal forma que o sistema possa refletir sobre a computação que ocorre no nível base. As arquiteturas reflexivas provêem um protocolo de meta-objetos[6], que estabelece o modo como os objetos presentes no nível base e os presentes no meta-nível estão relacionados.

[6]Em inglês: *Meta-Object Protocol (MOP).*

- Padrões de projeto. Possuem uma descrição mais refinada dos componentes lógicos e seus relacionamentos. Neste nível, os mecanismos de cooperação entre componentes são descritos e definidos para encontrar soluções de projeto em um dado domínio ou contexto. Geralmente a escolha de um padrão de projeto é influenciada pelos estilos/padrões arquiteturais escolhidos anteriormente. Padrões de projeto refinam as decisões de projeto definidas na fase do projeto arquitetural.

- Idiomas. São padrões de mais baixo nível, específicos para uma linguagem de programação. Os idiomas utilizam-se de características de uma determinada linguagem para descrever como implementar um aspecto particular de um componente do sistema ou do relacionamento entre eles.

O principal benefício decorrente do uso de padrões de software é facilitar a comunicação entre desenvolvedores de software, de uma mesma equipe ou independentes, ao permitir o emprego de estruturas de um nível de abstração maior do que linguagens de programação porém com o mesmo grau de formalismo dessas, contribuindo assim, de maneira positiva, para a reutilização de software.

3.4 *Frameworks* orientados a objetos

Arquiteturas de software definem e especificam, de maneira abstrata, a estrutura da colaboração de um conjunto de componentes lógicos que formam o sistema. *Frameworks* implementam parcialmente tal arquitetura, na forma de uma coleção de componentes de software ou classes concretas que podem ser adaptados e/ou instanciados por um desenvolvedor de aplicações. *Frameworks* orientados a objetos são blocos de construção bem maiores do que objetos e classes, que definem coleções de classes que permitem a reutilização em larga escala de ambos projeto e código [47]. Mais especificamente, um *framework* orientado a objetos é um conjunto de classes (abstratas e concretas) que provê uma infra-estrutura genérica de soluções para um conjunto de problemas específico. Padrões de projeto podem ser utilizados na construção de *frameworks* para facilitar a reutilização da arquitetura de software implementada pelo *framework*.

A infra-estrutura provida pelo *framework* pode ser definida em pontos adaptáveis e pontos fixos. Os pontos fixos realizam a estrutura geral e a lógica do domínio enquanto os pontos adaptáveis permitem que desenvolvedores de aplicações customizem o *framework* de tal forma que se adeque às características específicas de uma aplicação. Um *framework* é portanto visto como uma arquitetura de software semi-acabada para um domínio de problema que pode ser adaptado para resolver problemas específicos deste domínio.

Frameworks são quase que exclusivamente descritos em termos dos principais conceitos de orientação a objetos, como classes, objetos, herança e poliformismo [75, 86]. Dessa maneira, um *framework* orientado a objetos é um conjunto de classes que são integradas e interagem de uma maneira bem definida para fornecer um conjunto de serviços que satisfaça uma solução total ou parcial para um problema. *Frameworks* podem ser classificados de acordo com o modo em que a extensibilidade é obtida:

- *Framework* **caixa branca:** baseiam-se fortemente em mecanismos orientados a objetos, como por exemplo, herança e acoplamento dinâmico, para alcançar sua extensibilidade, provendo um conjunto de classes abstratas que são incompletas e que atuam como os pontos adaptáveis. Dessa maneira, um desenvolvedor de aplicações pode derivar classes específicas da sua aplicação a partir das classes abstratas definidas no *framework*, estendendo e sobrecarregando os métodos herdados.

- *Framework* **caixa preta:** contém todas as possíveis alternativas para todos os seus pontos adaptáveis. Uma aplicação é desenvolvida a partir de um *framework* caixa preta, escolhendo-se para cada ocorrência de um ponto adaptável, uma das alternativas disponíveis. Neste caso, ao contrário dos *frameworks* caixa branca, o usuário do *framework* necessita de pouco conhecimento sobre sua estrutura interna. Porém, os *frameworks* caixa preta são mais difíceis de serem desenvolvidos, pois requerem a definição antecipada de todas as formas possíveis de utilização dos seus pontos adaptáveis.

Ao contrário das abordagens tradicionais para reutilização de software que se limitam basicamente à construção de bibliotecas de classes, *frameworks* permitem reutilizar não apenas componentes isolados mas toda a arquitetura de software projetada para um domínio específico. A Figura 3.3 mostra as diferenças entre *frameworks* e bibliotecas de classes convencionais. *Frameworks* exibem uma inversão de controle em tempo de execução. Aplicações baseadas em bibliotecas de classes possuem o controle (i.e., o fluxo de eventos) localizado na própria aplicação. O programador da aplicação é encarregado de projetar/implementar o fluxo de controle da aplicação específica. Em contrapartida, aplicações baseadas em *frameworks* reutilizam o fluxo de eventos já embutido no próprio *framework*. Os programadores que utilizam um *framework* para construir a sua aplicação específica implementam apenas o código que será chamado pelo *framework* (*callbacks*). Desta forma, aplicações podem reutilizar o fluxo de eventos e a arquitetura de software que o *framework* disponibiliza. Enquanto que os componentes de uma biblioteca de classes são utilizados individualmente, os componentes de um *framework* são reutilizados como um todo para resolver uma instância específica de um certo conjunto de problemas relacionados.

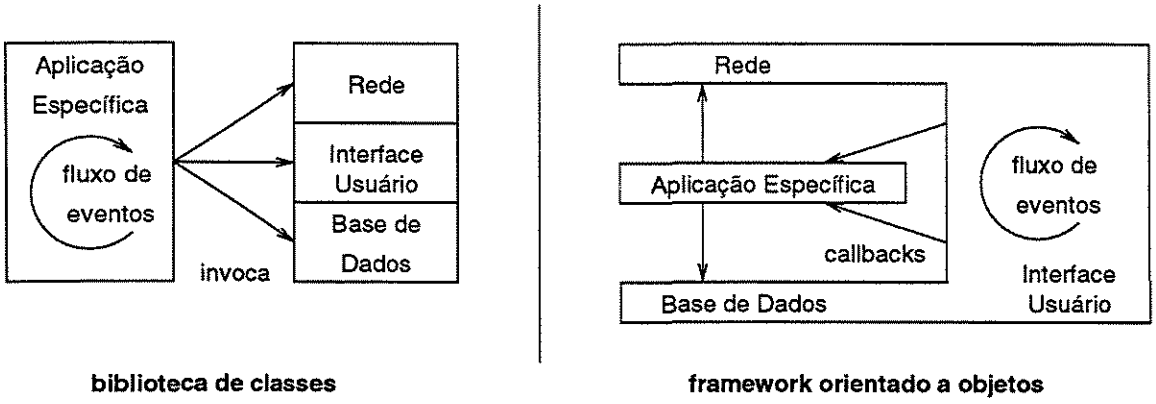


Figura 3.3: Diferenças entre bibliotecas de classes e *frameworks*

Dessa maneira, a construção de *frameworks* tornou-se popular como um veículo para reutilização. Eles oferecem a base sobre a qual um sistema pode ser construído. Utilizando o modelo de objetos, o desenvolvedor prossegue especializando, instanciando ou reutilizando as classes disponibilizadas, respeitando os relacionamentos já definidos. Uma vantagem derivada disso é que o processo de criação de uma aplicação torna-se fácil e demanda menos conhecimento específico do domínio da aplicação. Dessa maneira, o desenvolvedor necessita apenas saber como usar o *framework*, não requerendo conhecimento detalhado da complexidade abrangida pelo *framework*. Assim, pouco código terá que ser projetado e implementado. Somente será necessário o código adicional que corresponde às particularidades da aplicação a ser desenvolvida.

3.5 Componentes de software

Uma maneira de aumentar a produtividade no processo de desenvolvimento de software é a reutilização de software existente. O desenvolvimento de software baseado em componentes pode ser visto como uma abordagem de desenvolvimento que tem por objetivo a construção e a reutilização de componentes de software. O desenvolvimento de software baseado em componentes permite a construção de sistemas através da reutilização de componentes de software que já foram bem especificados e testados, diminuindo os custos do processo de desenvolvimento e aumentando a produtividade e a qualidade do software [40, 122]. Além disso, em um software desenvolvido a partir de componentes de software, é mais fácil realizar atividades de manutenção que objetivam adequar o software às mudanças impostas pelo ambiente no qual está inserido.

Os componentes de software seguem o estilo definido pelo modelo de objetos, onde existe a combinação das funções e dos dados relacionados em um único elemento.

Na realidade, os princípios fundamentais sobre componentes de software são os mesmos princípios que formam a base do modelo de objetos, como a unificação de dados e funções (operações), encapsulamento, onde um cliente de um objeto não tem conhecimento acerca de como as operações são implementadas, e identidade única, pois cada objeto possui uma única identidade que o representa independente de seu estado.

A definição de componentes de software estende os princípios do modelo de objetos, representando a noção da especificação de um objeto, de maneira explícita, através de interfaces. Com essa representação explícita obtém-se um nível de indireção entre o cliente de um componente de software e as capacidades deste componente de software. A especificação de todas as capacidades de um componente de software pode ser representado através de diversas interfaces. Com esse conceito, o desenvolvimento de software baseado em componentes diferencia-se de outras técnicas pela separação entre a especificação e a implementação de um componente de software, e também pela divisão da especificação em várias interfaces.

A separação da especificação em diferentes interfaces, permite que as dependências entre os componentes sejam restritas às interfaces específicas e não ao componente de software como um todo. Com essa separação, diminui-se o impacto e as dificuldades de modificações no sistema, visto que um componente de software pode ser trocado por outro, contanto que esse último especifique as mesmas interfaces do componente original. Essas características permitem que um componente de software seja atualizado ou trocado, sem impacto sobre os clientes desse componente.

A utilização de componentes de software pode levar ao problema de incompatibilidade arquitetural[7], onde esses componentes não atendem de maneira completa aos requisitos impostos pela arquitetura de software. O desenvolvimento de software através da integração de componentes de software existentes pode ser complicado pelos seguintes fatores: introduzir novos ou estender os requisitos de um componente (por exemplo, aumentar a confiabilidade de seus serviços), usar os componentes em um diferente contexto e heterogeneidade de componentes. Dessa forma, técnicas de adaptação de componentes são necessárias para contornar essas incompatibilidades [23, 129].

3.6 Considerações finais

Neste capítulo introduzimos as técnicas de engenharia de software para estruturação de software utilizadas nesse trabalho. Acreditamos que o modelo de objetos apresenta-se como um modelo promissor para o desenvolvimento de software confiável devido as características inerentes ao próprio modelo, tais como, abstração de dados, encapsulamento,

[7] Em inglês: *architectural mismatch*.

herança e reutilização.

Uma das vantagens deste modelo é o seu suporte à reusabilidade. Geralmente a idéia de reusabilidade estava associada apenas com reutilização de código. Desta forma, a reutilização de software era realizada apenas na fase de implementação do sistema. A noção de estilos arquiteturais (Seção 3.2) provê meios de explorar as semelhanças entre sistemas. Isto é, proporciona meios de reutilizar o conhecimento prévio sobre sistemas relacionados na definição da arquitetura de software de um sistema. Por outro lado, padrões de projeto (Seção 3.3) documentam conhecimentos e experiências de projetos existentes no intuito de auxiliar na busca de soluções apropriadas para problemas de projeto. *Frameworks* orientados a objetos (Seção 3.4) definem coleções de classes que permitem a reutilização em larga escala em ambos projeto e código. E por fim, o desenvolvimento de software baseado em componentes (Seção 3.5) permite a construção de software através da reutilização de componentes de software que já foram bem especificados e testados, diminuindo os custos do processo de desenvolvimento e aumentando a produtividade e a qualidade do software.

Capítulo 4 apresenta três estilos arquiteturais que documentam a arquitetura de software proposta para o desenvolvimento de sistemas orientados a objetos confiáveis. Capítulo 5 apresenta um conjunto coeso de padrões de projeto que provê soluções para o projeto dos elementos arquiteturais da arquitetura de software proposta. Ademais, este capítulo descreve um *framework* orientado a objetos que provê uma infra-estrutura para o desenvolvimento de sistemas confiáveis.

Capítulo 4

Uma Arquitetura de Software para Tolerância a Falhas

Nós alegamos que se as técnicas de tolerância a falhas discutidas no Capítulo 2 fossem documentadas de tal forma que facilitassem o seu entendimento e sua utilização, a construção de sistemas confiáveis seria bastante facilitada. Neste contexto, este capítulo apresenta uma arquitetura de software genérica para introduzir atomicidade, redundância de software, tratamento de exceções e recuperação de erros coordenada no desenvolvimento de sistemas confiáveis.

A arquitetura de software é baseada em três estilos arquiteturais: (i) componente tolerante a falhas ideal, (ii) colaborações entre papéis de componentes e (iii) reflexão computacional.

4.1 O estilo arquitetural componente tolerante a falhas ideal

Este estilo é uma especialização do estilo arquitetural **camada** [108], que define uma estrutura hierárquica do sistema em que cada camada fornece serviços para a camada superior e usa os serviços da camada inferior na hierarquia. Existem apenas duas formas de comunicação entre componentes neste estilo: requisições de serviços e respostas de serviços.

4.1.1 Componentes e conectores

Os componentes lógicos deste estilo são componentes tolerante a falhas ideais [74] que possuem duas portas diferentes (*requisita* ou *provê*). O conector definido por este estilo define dois tipos de componentes tolerante a falhas ideais:

- Componentes que recebem requisições de serviços e produzem respostas (via uma porta *provê*). Se o serviço é realizado de acordo com sua especificação, o componente retorna uma resposta normal, caso contrário ele retorna uma resposta anormal (excepcional). As respostas normais e excepcionais estão definidas na interface do componente, então clientes de seus serviços devem possuir meios de lidar com ambas as respostas.

- Componentes que requisitam serviços para outros componentes (via uma porta *requisita*). Este componente deveria ser apto para lidar com as respostas (normais e excepcionais) retornadas.

Figura 4.1 ilustra a organização estrutural deste estilo arquitetural. Um componente tolerante a falhas ideal é dividido em duas partes, a parte normal que implementa as atividades normais do componente, e a parte anormal que implementa as medidas para tolerar as falhas que causam respostas excepcionais. Em uma situação ideal, componentes interagem apenas produzindo respostas normais. Entretanto, considerando que o sistema não é livre de falhas, exceções podem ser produzidas como respostas para requisições de serviços.

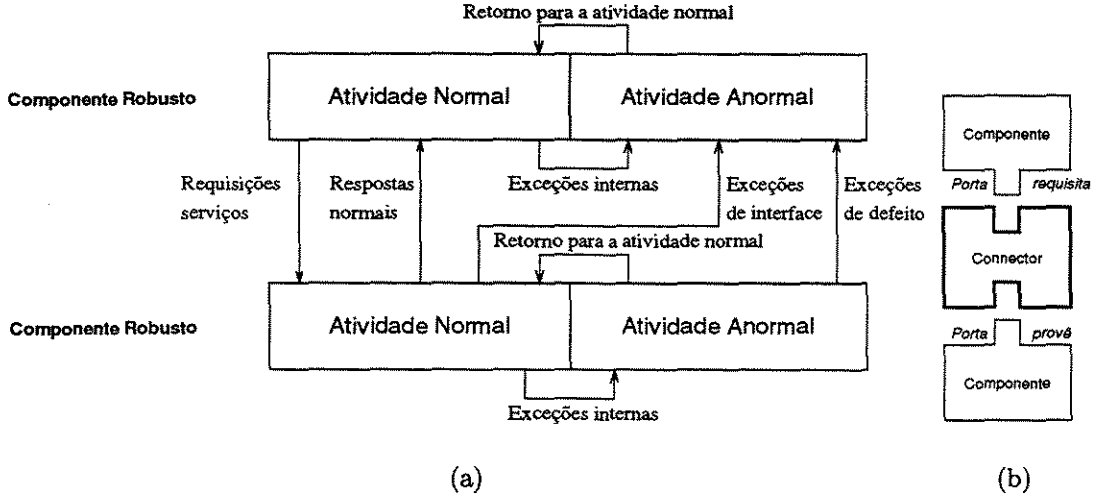


Figura 4.1: O estilo arquitetural componente tolerante a falhas ideal

As respostas excepcionais podem ser classificadas em:

- Exceções de interface: são sinalizadas em resposta às requisições que não satisfazem a especificação da interface do componente. Por exemplo, o valor de um parâmetro está fora de um limite especificado. Geralmente, estas exceções são levantadas quando as precondições de um serviço não são satisfeitas.

- Exceções locais ou internas: são exceções geradas pelo componente quando este detecta alguma condição anormal interna.
- Exceções de defeito: são sinalizadas se um componente determina que por alguma razão ele não pode prover os seus serviços. Geralmente, estas exceções são levantadas quando as pós-condições de um serviço não são satisfeitas.

Se estas exceções são tratadas adequadamente (isto é, o componente foi apto a mascarar esta exceção), o componente pode retornar a prover os seus serviços normais. Entretanto, caso o componente não tenha sucesso no tratamento destas exceções, ele deveria propagar uma exceção de defeito.

Ademais, componentes tolerante a falhas ideais podem ser projetados utilizando diversidade de projeto (Seção 2.1.4). Isto é, ele pode ser projetado como um componente que contém uma série de variantes que implementam as mesmas funcionalidades mas utilizando projetos alternativos.

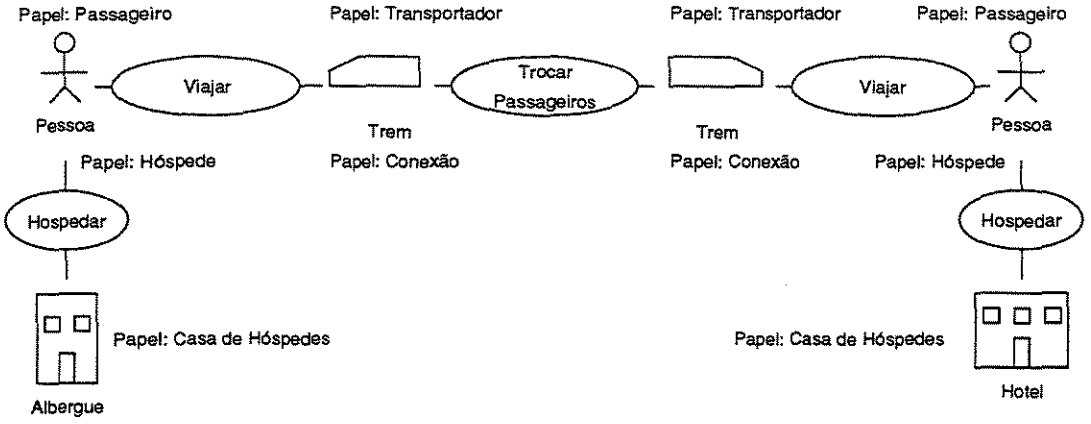
4.2 O estilo arquitetural colaborações entre papéis

Sistemas de software são representados como uma composição de colaborações, em descrições que usam o projeto baseado em colaborações como base. Segundo a terminologia adotada pela comunidade de projeto baseado em colaborações, um papel de componente é aquela porção da especificação do componente que descreve a interação deste componente no contexto de uma colaboração que descreve um conjunto de atividades que determinam como estes componentes interagem [40, 112, 128].

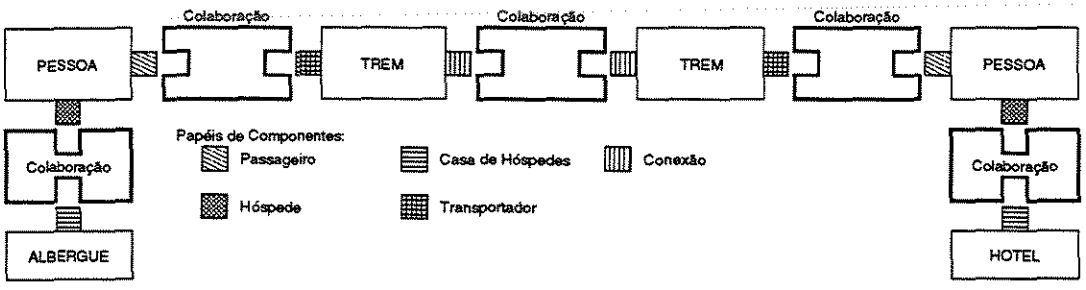
Componentes podem participar de diversas interações (colaborações), desempenhando diferentes papéis em cada uma destas. Por exemplo (Figura 4.2), uma pessoa que se hospeda em um hotel desempenha o papel de hóspede em sua interação com a recepção do hotel. A mesma pessoa quando viajando para a cidade onde se localiza o hotel, desempenha o papel de passageiro na sua interação com a companhia ferroviária. Além disso, os passageiros podem fazer conexões (troca de trem) com o objetivo de tornar a sua viagem mais rápida.

4.2.1 Componentes e conectores

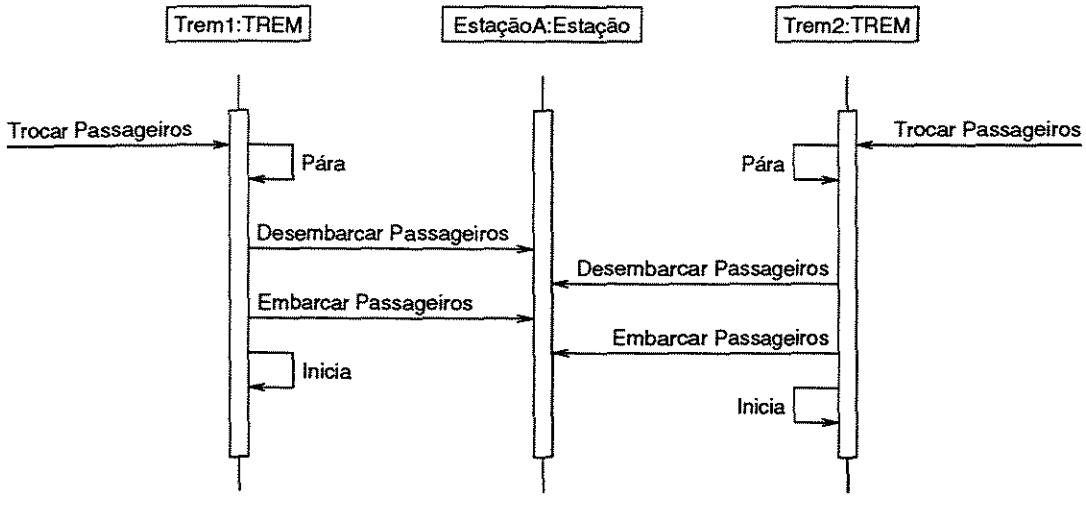
Os papéis de componentes são os componentes lógicos deste estilo arquitetural enquanto que as colaborações são os conectores. A mesma especificação de uma colaboração se aplica para diferentes conjuntos de componentes, desde que os correspondentes componentes desempenhem os mesmos papéis. Por exemplo, se pessoa deseja economizar dinheiro, a



(a)



(b)



(c)

Figura 4.2: O estilo arquitetural colaborações entre papéis

pessoa poderia decidir se hospedar em um albergue da juventude e viajar de ônibus. Neste caso, a pessoa desempenha os mesmos papéis, apesar de que os outros papéis (transportador e casa de hóspedes) são desempenhados por outros componentes (ônibus e albergue da juventude).

Figura 4.2(c) ilustra o diagrama de seqüência que representa a colaboração Trocar Passageiros. Ao chegar próximo de uma estação ferroviária, cada trem deve realizar a seguinte seqüência de operações: parar em uma plataforma da estação ferroviária, desembarcar alguns passageiros nesta plataforma, embarcar outros e por fim, sair da estação ferroviária.

Tipos diferentes de colaborações

A noção de colaboração é bem aceita na comunidade de projeto orientado a objetos. Entretanto, existem diversas aplicações em que apenas a noção de colaboração não é suficiente para representar o comportamento cooperativo entre componentes. Em complexas aplicações concorrentes é também necessário capturar a noção de coordenação no suporte ao tratamento de erros e recuperação coordenada entre múltiplos componentes. Por exemplo, existe a possibilidade de acontecerem diferentes situações excepcionais durante a execução da colaboração TrocarPassageiros tal como, um trem não consegue parar (falha no freio). Desta forma, este estilo define diferentes tipos de conectores (colaborações) que apresentam características que facilitam o projeto de aplicações que são supostas a lidar com falhas. Isto é, esta colaboração determina a seqüência de atividades que os trens realizam e como estes trens interagem.

Figura 4.3 ilustra os diferentes conectores definidos por este estilo:

- O primeiro tipo de conector define uma colaboração simples [112]. Este conector não apresenta características relacionadas à provisão de características de tolerância a falhas.

- Se foi determinado que não existe interação entre aquele grupo de componentes e o restante do sistema durante a atividade cooperativa, então esta atividade deveria ser especificada como uma ação atômica [74]. Ações atômicas (Seção 2.2.2) provêem confinamento do prejuízo dado que eles garantem que nenhuma informação errônea ultrapassa as barreiras definidas pela ação atômica.

- Se colaborações supostamente lidam com falhas, então é interessante incorporar explicitamente a noção de coordenação no suporte à recuperação coordenada entre múltiplos componentes. Além disso, diferentes técnicas de recuperação de erros podem ser utilizadas: apenas recuperação por retrocesso [97, 100], apenas recuperação por avanço (tratamento de exceções) [51, 52, 53] ou uma combinação destas [27].

- E por fim, se colaborações devem lidar com ambos os tipos de concorrência [61] (competição e cooperação), então a colaboração deveria implementar recuperação de erros coordenada e manter a consistência de recursos externos [130] na presença de falhas e concorrência entre diferentes atividades cooperativas competindo por estes recursos.

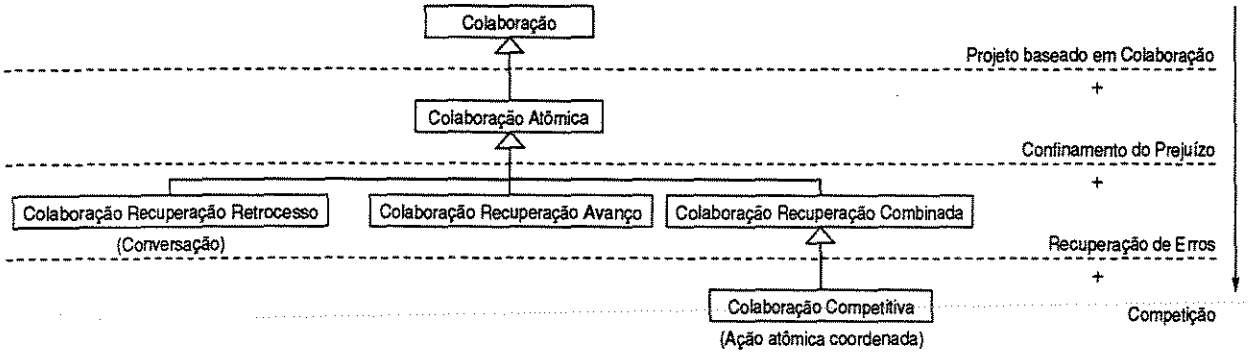


Figura 4.3: Diferentes tipos de colaborações

4.3 O estilo arquitetural meta-nível

Arquiteturas de meta-níveis são baseadas no conceito de reflexão computacional que é uma técnica que permite a um sistema manter informação sobre si mesmo (meta-informação) e usar esta informação para alterar o seu comportamento [84, 113]. Isto significa que um componente pode obter informações sobre as propriedades internas de outro componente e baseado nessas informações dinamicamente modificar as suas computações.

De maneira geral, uma aplicação depende de dois tipos distintos de propriedades: propriedades funcionais, associadas ao propósito da aplicação e propriedades não-funcionais, relacionadas com a infra-estrutura de apoio do sistema para realização deste propósito. Com domínios de atuação tão diferentes era de se esperar que uma aplicação bem modularizada apresentasse uma separação nítida entre estes requisitos. O uso de técnicas orientadas a objetos, como encapsulamento, melhoram a modularidade da aplicação mas não são suficientes para garantir tal separação. Esta modularidade, entretanto, é facilmente obtida utilizando-se a técnica de reflexão computacional que modulariza o sistema em pelo menos dois níveis: o nível base e o meta-nível. No meta-nível estão os meta-objetos que lidam com o processamento das meta-informações e gerenciamento das aplicações, enquanto no nível base estão os objetos responsáveis pela implementação das propriedades funcionais das aplicações. Arquiteturas de meta-níveis provêem meios de implementar

requisitos não-funcionais, tais como tolerância a falhas [26, 43, 76], persistência [96, 118] e distribuição [117, 134], de forma transparente aos requisitos funcionais da aplicação. Meta-objetos são responsáveis pela implementação de tais requisitos não-funcionais. Um meta-objeto pode ser controlado por um outro meta-objeto que se localiza em um nível acima (meta-meta-nível). Este processo pode ser recursivo, originando um torre recursiva de meta-níveis.

Para o meta-nível ser apto a refletir sobre objetos do nível base, ele deveria ter informações relacionadas à estrutura destes objetos (meta-informação estrutural). Além disso, interações entre objetos deveria ser materializada em objetos de tal forma que meta-objetos pudessem inspecioná-las e possivelmente alterá-las. Isto é alcançado por interceptar operações no nível base (por exemplo, invocações de métodos) criar objetos que materializem estas operações, e transferir o fluxo de controle para o meta-nível. Este processo é chamado materialização[1]. Após transferir o controle para o meta-nível, os meta-objetos podem inspecionar a informação materializada e podem também modificar os aspectos estruturais e/ou dinâmicos dos objetos presentes no nível base. Este processo é chamado de reflexão [84, 44].

4.3.1 Componentes e conectores

O conector deste estilo é o protocolo de meta-objetos[2] (MOP), que estabelece o relacionamento entre os objetos e meta-objetos (componentes lógicos) presentes no nível e no meta-nível respectivamente. O MOP provê uma interface para a linguagem de programação que revela informações normalmente escondidas pelo compilador e/ou ambiente de execução. Figura 4.4 ilustra a organização estrutural de um sistema que é organizado de acordo com este estilo arquitetural. As interações entre os objetos e meta-objetos são realizados pelo protocolo de meta-objetos que estabelece as diretrizes que guiam a construção de um sistema organizado por este estilo. Com propósitos de ilustração, suponha que para cada Objeto presente no nível base, existe um correspondente Meta-Objeto que representa os aspectos estruturais e dinâmicos de Objeto. Como ilustrado na Figura 4.4, se um Cliente envia uma requisição de serviço para Objeto, o Meta-Objeto intercepta esta requisição, materializa a computação do nível base e retorna o resultado para Cliente. Sobre o ponto de vista de Cliente, reflexão computacional é transparente: Cliente envia uma requisição de serviço e recebe o resultado sem ter conhecimento que a requisição foi interceptada e direcionada para o meta-nível.

Com o objetivo de obter uma composição estruturada de requisitos não-funcionais, este estilo define outro tipo de conector que determina as regras de interação entre os meta-

[1] Em inglês: *reification*.
[2] Em inglês: *Meta-Object Protocol*.

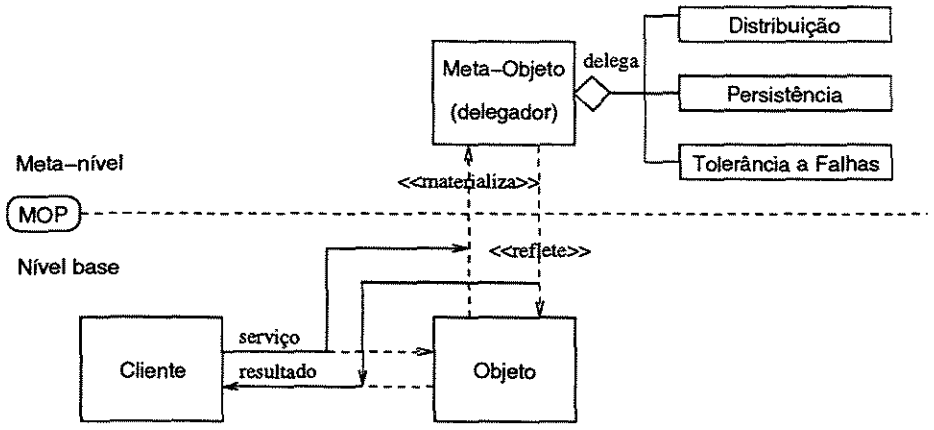
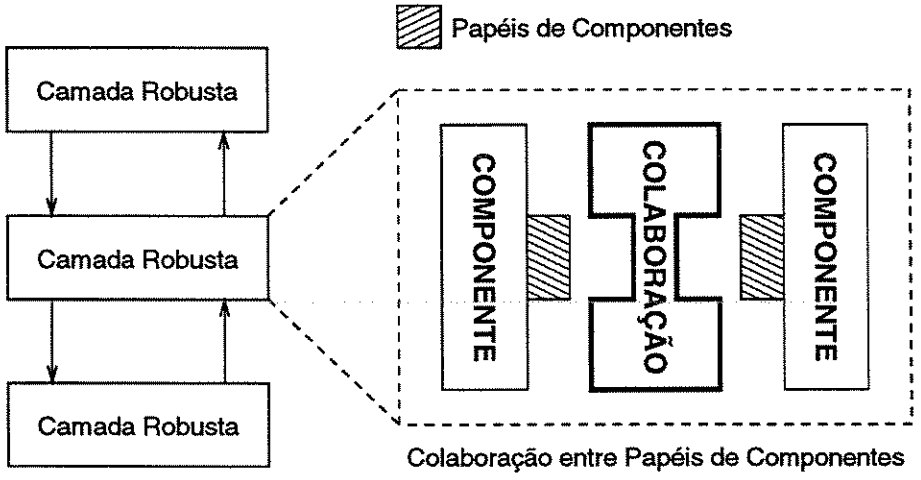


Figura 4.4: Uma arquitetura de meta-níveis

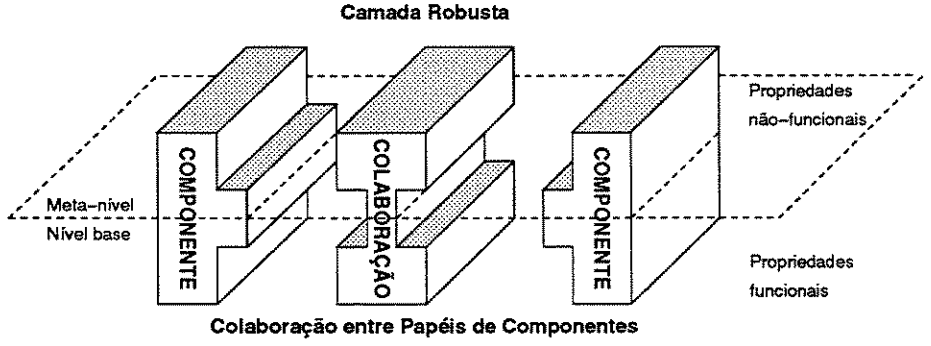
objetos que implementam as propriedades não-funcionais. Este conector define dois tipos de meta-objetos: delegadores e delegados. Delegadores delegam operações e resultados para outros meta-objetos. Desta forma, objetos presentes no nível base podem estar associados a um meta-objeto que encapsula (compõe) diversas propriedades não-funcionais. A semântica de tais composições é definida por este conector.

4.4 A arquitetura de software proposta

Nesta seção, nós apresentamos uma arquitetura de software para o desenvolvimento de aplicações concorrentes confiáveis que é baseada principalmente na combinação dos estilos arquiteturais: componente tolerante a falhas ideal (Seção 4.1), colaborações entre papéis (Seção 4.2) e meta-nível (Seção 4.3). Nós usamos uma estrutura em camadas onde cada camada provê serviços para a camada superior e é cliente da camada inferior da hierarquia. Ademais, componentes tolerante a falhas ideais presentes na mesma camada podem realizar atividades cooperativas, cada uma desempenhando um específico papel de componente em uma colaboração. Desta forma, é muito importante garantir a consistência destes componentes dado que cada componente pode simultaneamente prover diversos serviços a diferentes clientes e está envolvido em diversas colaborações. Em adição, meta-objetos são responsáveis pela implementação das propriedades não-funcionais dos componentes tolerante a falhas ideais (Figura 4.5).



(a) Visão bidimensional da arquitetura



(b) Visão tridimensional de uma camada

Figura 4.5: A arquitetura de software proposta

Capítulo 5

Padrões de Projeto para Tolerância a Falhas

Padrões de projeto refinam as decisões definidas na fase do projeto arquitetural. Neste capítulo, apresentamos um sistema de padrões que define um conjunto coeso de padrões que refinam os componentes lógicos e conectores da arquitetura de software para tolerância a falhas proposta no Capítulo 4 e que auxiliam o desenvolvimento de sistemas confiáveis. Os padrões apresentados neste capítulo são os seguintes:

1. padrões que provêem soluções de projeto para a implementação dos componentes lógicos do estilo arquitetural componente tolerante a falhas ideal (seção 4.1).

2. padrões que provêem soluções de projeto para a implementação dos conectores do estilo arquitetural componente tolerante a falhas ideal (seção 4.1).

3. padrões que provêem soluções de projeto para a implementação dos componentes lógicos do estilo arquitetural colaborações entre papéis (seção 4.2).

4. padrões que provêem soluções de projeto para a implementação dos conectores do estilo arquitetural colaborações entre papéis (seção 4.2).

A descrição dos padrões é baseada no formato proposto por Gamma *et al.* [49] que identifica os seguintes elementos:

- O **problema** que ocorre em um determinado **contexto**, incluindo restrições e exigências que devem ser atendidas pela solução.

- A **solução** detalhada do problema. A **estrutura** do projeto da solução descreve os participantes (objetos e classes) e suas responsabilidades. Esta estrutura é uma espécie de modelo que pode ser seguido em várias situações diferentes. A **dinâmica** da solução descreve o relacionamento dinâmico entre os participantes.

- As **variantes** que descrevem possíveis variações na estrutura do projeto da solução.
- Os **usos conhecidos** destes padrões encontrados na literatura.
- As **conseqüências** que descrevem as vantagens e desvantagens do emprego do padrão de projeto.

5.1 Componente tolerante a falhas ideal

Nesta seção apresentamos um conjunto de padrões que provêem soluções de projeto para a implementação dos componentes lógicos do estilo arquitetural componente tolerante a falhas ideal (Seção 4.1).

5.1.1 Padrão contrato reflexivo

Contexto

Confiabilidade é um requisito básico para qualquer sistema de software. Infelizmente, nenhum sistema de software pode ser assumido estar livre de falhas. Mesmo empregando testes e uma depuração cuidadosa ou usar métodos formais não garante que o sistema estará livre de falhas. Desta forma, sistemas computacionais devem esperar a ocorrência de falhas e tratá-las com mecanismos apropriados.

Problema

O desafio está em como detectar erros durante o tempo de execução para permitir o sistema tratar tais situações. As seguintes forças estão relacionadas a este problema de projeto:

- Um erro pode apenas ser detectado em relação a especificação do correto comportamento do sistema. A especificação é assumida ser correta. Neste contexto, detecção de erros significa verificar o estado e o comportamento do sistema em relação a sua especificação (Seção 2.2.1).
- Para facilitar o entendimento e manutenção dos componentes da aplicação, deveríamos evitar a incorporação desordenada do conjunto de detectores de erros. Idealmente, a incorporação de detectores de erros deveria ser feita de forma não-intrusiva aos componentes da aplicação.

- Para permitir flexibilidade, deveríamos prover uma forma de ativar e desativar os detectores de erros.

Solução

A especificação de um componente pode ser feita utilizando a metodologia de projeto por contrato introduzida por Meyer [88]. Um contrato define os aspectos observáveis pelos clientes de um componente de software. Contratos são definidos através de precondições, pós-condições e invariantes. Se uma destas condições é violada, então uma exceção é levantada. Um contrato especifica obrigações e direitos dos contratuentes: o cliente e o servidor do serviço. Ele especifica quais são as restrições sobre os parâmetros de entrada de um serviço que devem ser satisfeitas pelo cliente, ou seja, as precondições, e qual é o comportamento que pode ser esperado pelo cliente na execução de um serviço, caso estas restrições sejam atendidas, ou seja as pós-condições. Além disso, especifica as invariantes, que são todas as restrições impostas sobre o estado interno do componente.

A metodologia de projeto por contrato tem como suporte a linguagem Eiffel [89, 124], que implementa a verificação de precondições, pós-condições e invariantes de classes embutida na própria linguagem. A dificuldade consiste em fazer o mapeamento dos contratos definidos na especificação para o projeto de componentes, considerando-se linguagens que não fornecem este tipo de suporte.

Com objetivo de separar a implementação dos serviços funcionais do componente, das atividades que checam as precondições, pós-condições e invariantes dos serviços deste componente, nós utilizamos a técnica de reflexão computacional (Seção 4.3).

Estrutura

O padrão **contrato reflexivo** [10] define três tipos de elementos: (i) classes normais, (ii) classes contratos e (iii) a meta-classe MetaContract. As classes normais estão presentes no nível base e implementam as atividades normais da aplicação. As classes contratos estão presentes no nível base e definem métodos que implementam os testes de invariantes, precondições e pós-condições para cada serviço crítico do componente (classe normal), retornando exceções caso alguma destas condições não seja satisfeita. Nas classes contratos, para cada serviço definido no componente (classe normal), existe um método correspondente que verifica as precondições, testando os parâmetros de entrada, podendo também testar condições relacionadas com o estado interno do componente antes de um serviço ser executado (Figura 5.1). De forma análoga, para cada serviço do componente, existe um método que testa as pós-condições que devem ser satisfeitas pelo resultado do serviço, podendo testar também o estado interno do componente depois que o serviço é executado.

Além disso, existe um método que testa as invariantes do componente. Todos os métodos das classes contrato devem receber como parâmetro uma referência para o componente, para que possam fazer os testes sobre o estado interno do componente.

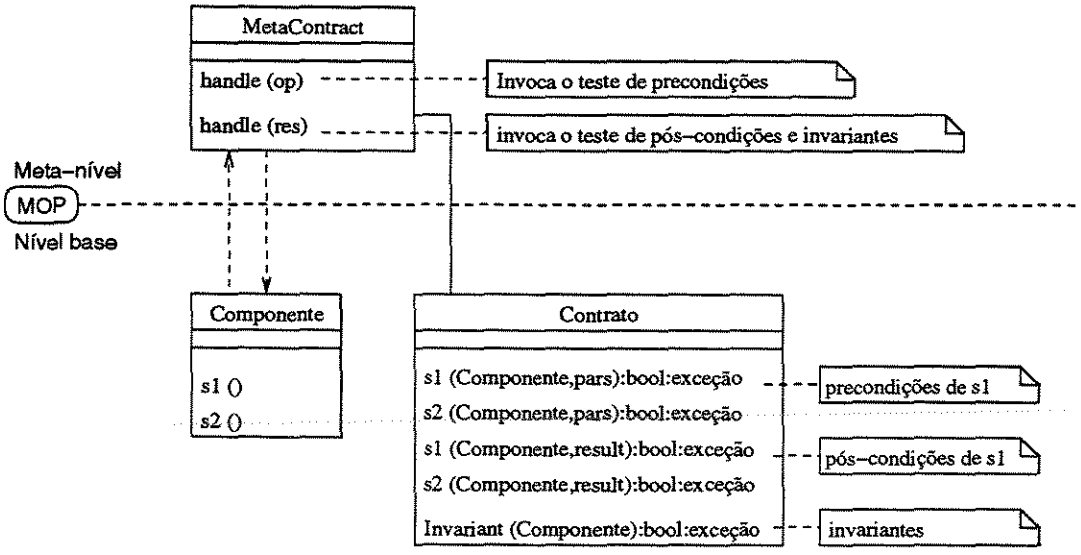


Figura 5.1: Padrão contrato reflexivo

Todos os métodos das classes contratos retornam como resultado um valor booleano verdadeiro, caso as condições sejam satisfeitas, ou retornam exceções, caso contrário. Os métodos que testam precondições retornam exceções que correspondem às exceções de interface do modelo do componente ideal tolerante a falhas (Seção 4.1). Os métodos que testam as pós-condições retornam exceções que correspondem às exceções de defeito dado que se o resultado do serviço não atende às pós-condições impostas pelo lado cliente do contrato, implica que o componente falhou em prover o serviço. O método que testa as invariantes do componente também retorna exceções de defeito, pois se as condições sobre o estado interno do componente são violadas, o componente não pode prover os seus serviços corretamente.

Instâncias da meta-classe MetaContract são responsáveis pela invocação dos métodos que verificam o contrato, de uma forma transparente para a classe normal presente no nível base. Para isto, utiliza o mecanismo de interceptação e materialização de mensagens provido pelo protocolo de meta-objetos. A meta-classe MetaContract (Figura 5.1) implementa dois métodos básicos: handle(op) que inspeciona a operação interceptada antes que esta seja executada e invoca os testes de precondições; e handle(res) que inspeciona o resultado da operação, e invoca os testes de pós-condições e invariantes do componente.

A classe normal pode ser estendida através de herança, e deve-se garantir que o contrato da subclasse honra o contrato da superclasse, não violando as condições impostas

neste último. Para isto, é necessário uma construção sistemática de classes de contratos e subcontratos, atendendo às seguintes regras [88]:

- Precondições: as precondições estabelecidas no contrato da superclasse podem ser relaxadas no subcontrato estabelecido pela subclasse. O subcontrato pode, portanto sobrecarregar os métodos que implementam as precondições, podendo diminuir restrições impostas sobre os parâmetros ou estado interno do componente, ou até mesmo eliminá-las.

- Pós-condições: as pós-condições estabelecidas no contrato da superclasse podem ser mais restritas no subcontrato estabelecido pela subclasse. Ou seja, no mínimo, as restrições impostas sobre o resultado dos métodos da superclasse são atendidas, podendo ainda ser mais fortes. Para que isto seja garantido, os métodos do subcontrato que implementam as pós-condições deveriam invocar o método de pós-condições do supercontrato, garantindo-se que estas pós-condições são atendidas.

- Invariante: as invariantes da superclasse devem sempre ser atendidas pelas instâncias da subclasse, podendo-se ainda estender as condições impostas com novas restrições sobre o estado interno de uma instância da subclasse. Portanto, o método do subcontrato que implementa a invariante da subclasse deveria sempre invocar o método de invariante no seu supercontrato, podendo acrescentar novas condições.

Dinâmica

O seguinte diagrama (Figura 5.2) ilustra um cenário típico deste padrão. Neste cenário, as precondições, pós-condições e invariantes foram satisfeitas:

1. O Cliente invoca s1 em Componente;

2. O MetaContract intercepta esta invocação de operação e checa as precondições desta operação (invocando o correspondente método de Contrato);

3. Se as precondições são satisfeitas, o MetaContract invoca s1 em Componente. Caso contrário, retorna uma exceção de interface para Cliente;

4. O MetaContract intercepta o resultado da operação e checa as pós-condições desta operação (invocando o correspondente método de Contrato). Caso contrário, retorna uma exceção de defeito para Cliente;

5. Se as pós-condições são satisfeitas, o MetaContract checa as invariantes deste componente (invocando o correspondente método de Contrato). Caso contrário, retorna uma exceção de defeito para Cliente;

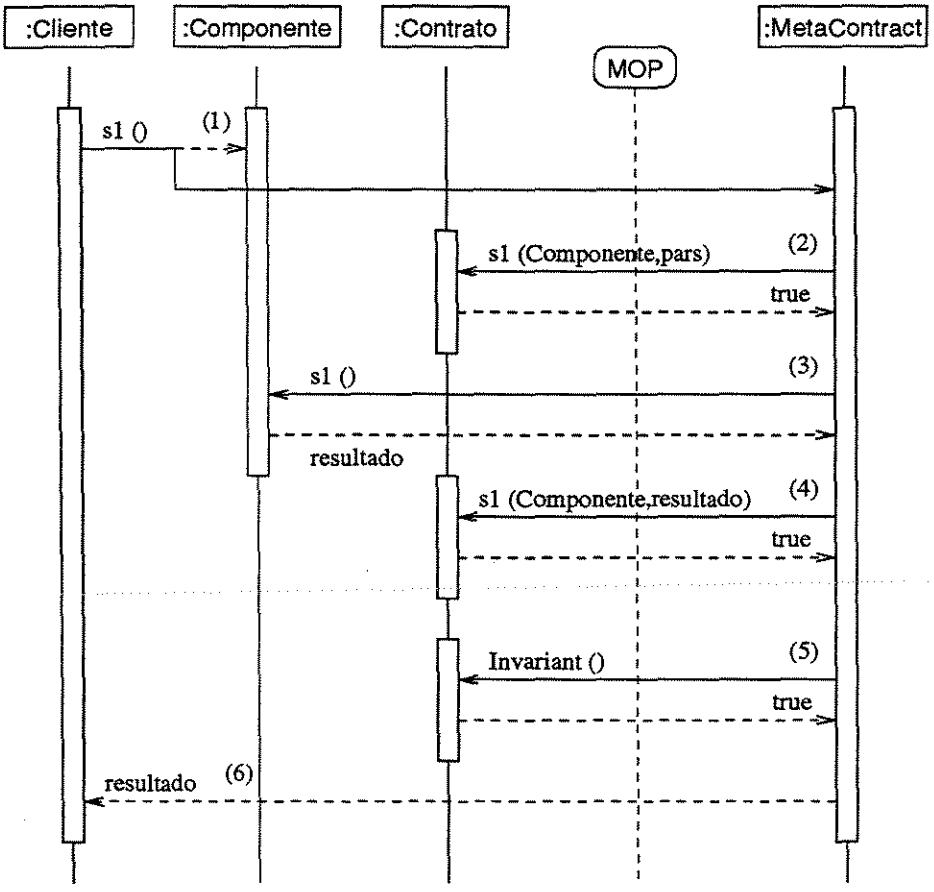


Figura 5.2: Dinâmica: padrão contrato reflexivo

6. Se as invariantes deste componente são satisfeitas, o MetaContract retorna o resultado da operação para Cliente.

Variantes

- **Variante totalmente reflexiva.** Nesta variante, a implementação das precondições, pós-condições e invariantes são implementados pelos próprios meta-objetos. Isto é, a classe contrato e a meta-classe MetaContract são unidas e tornam-se apenas uma única meta-classe.

- **Variante não reflexiva.** Nesta variante, a responsabilidade de invocar os testes de precondições, pós-condições e invariantes são transferidas para classes presentes no nível base. Apesar desta variante não necessitar de um protocolo de meta-objetos para implementar contratos em linguagens de programação orientadas a objetos, o código da aplicação (Componente) possui invocações explícitas para os métodos que

verificam o contrato.

Usos conhecidos

- Meyer [88] propôs a metodologia de projeto por contrato no contexto da linguagem Eiffel [89] que é uma linguagem orientada a objetos fortemente tipada que suporta o conceito de contratos. Com algumas extensões em C++ [41], Smalltalk [28] e Java [35], a metodologia projeto por contrato não é mais exclusiva a Eiffel. Estas extensões utilizam a variante não reflexiva discutida anteriormente.
- Tramontana e de Lemos [125] utiliza a variante totalmente reflexiva discutida anteriormente.

Conseqüências

- **Transparência.** O uso de reflexão computacional e dos mecanismos de interceptação e materialização de mensagens torna a implementação do contrato transparente para a classe normal do componente. O MetaContract funciona como uma extensão às linguagens de programação que não implementam os testes de invariante, precondições e pós-condições embutidos na própria linguagem, dando a ilusão de que estes testes são realizados automaticamente.
- **Manutenção.** Desde que a implementação do contrato é transparente à classe normal do componente, isto facilita as extensões ou modificações quanto às condições do contrato do componente, visto que a classe contrato pode ser alterada separadamente sem afetar a classe normal do componente.
- **Ausência de checagem.** Desde que a associação entre a classe normal do componente e o meta-objeto (instância de MetaContract) é dinâmica, pode-se fazer associações indesejadas, por exemplo, associar um componente a um meta-objeto que não implementa o seu contrato.
- **Subcontratos.** A definição de uma hierarquia de contratos, paralela à hierarquia de componentes, torna fácil a implementação direta das regras impostas para o subcontrato, que garantem a consistência de subtipo. Este padrão propõe uma forma sistemática para a redefinição dos métodos do contrato da superclasse na classe que implementa o subcontrato, que sugere o cumprimento das regras. Entretanto não existe garantia de que, seguindo estas regras, o subcontrato não viola o contrato da superclasse, visto que o relaxamento das precondições e a restrição das pós-condições dependem da implementação destas condições.

5.1.2 Padrão estratégia de tratamento de exceções

Contexto

Ocorrências de exceções podem ser detectadas durante a execução de uma região protegida da atividade normal da aplicação. O fluxo de controle normal é desviado para o código excepcional e um tratador apropriado é procurado.

Problema

A arquitetura de software deveria ser organizada de tal forma que os componentes responsáveis pelo desvio do fluxo de controle normal e pela procura do tratador realizem suas atividades gerenciais de forma não-intrusiva à aplicação. A seguinte força surge quando lidando com este problema:

- O modelo escolhido para a continuação do fluxo de controle deveria ser o modelo de terminação desde que é mais apropriado para desenvolver sistemas confiáveis (Seção 2.3.4).

Solução

Nós utilizamos a técnica de reflexão computacional com o objetivo de separar classes responsáveis pelas atividades de gerenciamento (meta-nível) daquelas que implementam as atividades normais da aplicação (nível base). O nível base define a lógica da aplicação onde classes normais implementam as atividades normais da aplicação. O meta-nível consiste de meta-objetos que realizam a busca transparente de tratadores de exceções. Meta-objetos estão associados a instâncias de classes normais, e mantém meta-informações relacionadas às regiões de proteção definidas no nível base. Uma região protegida pode ser um método, um objeto ou uma classe. O protocolo de meta-objetos é responsável pela interceptação dos resultados dos métodos e pela troca do fluxo normal para o excepcional quando exceções são detectadas, por transferir o controle para o meta-nível. Com a meta-informação disponível, meta-objetos conseguem encontrar o tratador que deveria ser invocado quando uma exceção é detectada em uma dada região protegida. Quando a execução dos tratadores é concluída, o protocolo de meta-objetos retorna para o fluxo normal da aplicação seguindo o modelo de terminação.

Estrutura

O padrão **estratégia de tratamento de exceções** (Figura 5.3) introduz dois tipos de elementos: (i) classes normais e (ii) meta-objetos do tipo MetaSearcher. As classes normais estão presentes no nível base e definem as atividades normais de uma aplicação específica. Eles estão associados a correspondentes classes excepcionais. Meta-objetos do tipo MetaSearcher estão associados a instâncias de classes normais, e são responsáveis pela interrupção do fluxo de controle normal e pela procura dos tratadores.

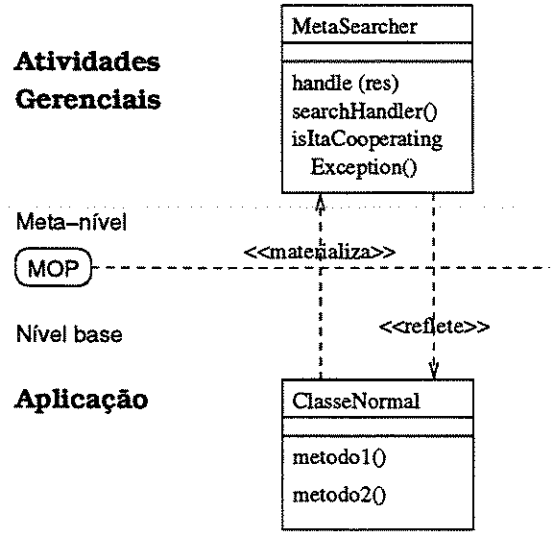


Figura 5.3: Padrão estratégia de tratamento de exceções

Dinâmica

O seguinte diagrama (Figura 5.4) ilustra um cenário típico deste padrão. Neste cenário, suponha que o método metodo1 da aplicação levante uma exceção e o tratador associado a esta exceção foi executado com sucesso.

1. Cliente invoca metodo1 em ClasseNormal;

2. Durante a execução de metodo1, ClasseNormal levanta a exceção Exception. A ocorrência da exceção é interceptada pelo meta-objeto (instância de MetaSearcher) que está associado a ClasseNormal;

3. O MetaSearcher solicita ao meta-objeto, instância de MetaHandler (Seção 5.1.3) associado ao tratador dessa exceção que inicie o seu tratamento;

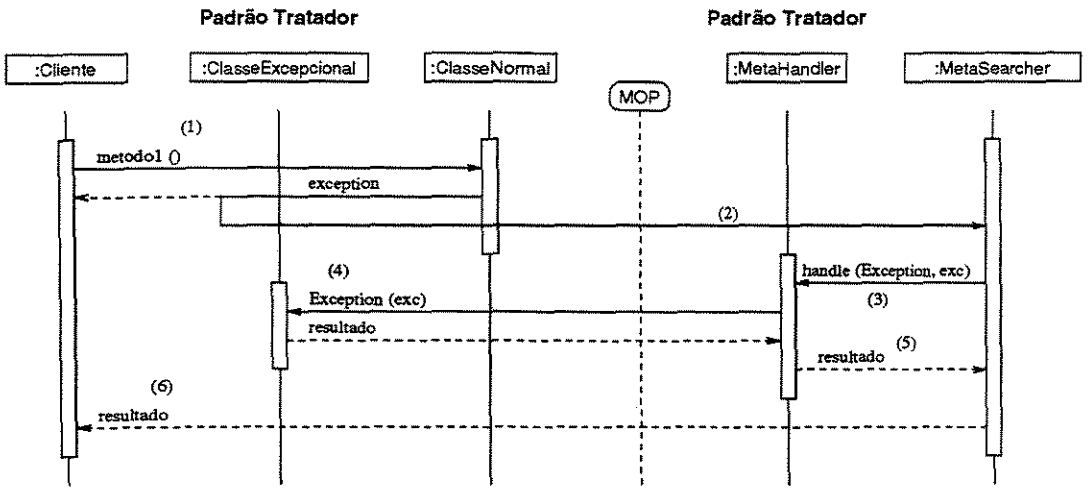


Figura 5.4: Dinâmica: padrão estratégia de tratamento de exceções

4. O MetaHandler invoca transparentemente o método de ClasseExcepcional que implementa o tratador para esta exceção;

5. O tratador da exceção, então retorna o resultado do tratamento da exceção para o MetaHandler que o devolve para MetaSearcher;

6. O MetaSearcher então devolve este resultado para o cliente.

Variantes

- **Tratadores reflexivos.** Esta variante [90] transfere os métodos que implementam os tratadores das classes excepcionais para o meta-nível. Os meta-objetos associados a instâncias de classes normais contém métodos responsáveis por realizar a recuperação de erros (tratamento de exceções). Ao invés de utilizar princípios reflexivos para separar completamente os aspectos funcionais dos mecanismos gerenciais da aplicação, esta variante explora reflexão computacional para separar o código normal e anormal.

- **Exceções materializadas.** Nesta variante [90], a exceção é a entidade materializada ao invés do resultado da operação. Tal abordagem permite que a exceção controle o seu tratamento. Conseqüentemente é possível implementar o modelo de continuação desde que o fluxo de controle é interrompido exatamente onde a exceção foi levantada.

Usos conhecidos

- Mitchel *et al.* [90] utiliza as variantes tratadores reflexivos e exceções materializadas discutidas anteriormente.
- Hof *et al.* [62] usa a técnica de reflexão computacional com o objetivo de obter informações relacionadas às regiões protegidas e os tratadores associados a estas.

Conseqüências

- **Transparência.** Os meta-objetos ligam transparentemente a atividade normal e os correspondentes tratadores de exceções de tal forma que programadores não precisam utilizar comandos especiais para especificar regiões protegidas.
- **Legibilidade.** O código normal não é "poluído" com métodos que realizam as atividades de recuperação de erros. Como conseqüência, ambos os códigos normais e excepcionais são fáceis de serem entendidos e mantidos.
- **Compatibilidade.** O padrão pode ser usado em conjunto com a estratégia de tratamento de exceções implementada na linguagem de programação utilizada, e estes podem se complementarem um ao outro.

5.1.3 Padrão tratador

Contexto

Projetistas de aplicações confiáveis necessitam especificar tratadores de exceções para as exceções locais e concorrentes que podem ocorrer durante a execução de suas aplicações. Um tratador é invocado quando uma exceção correspondente é levantada.

Problema

A infra-estrutura da arquitetura de software deveria ser organizada de tal forma que permita projetistas definirem os tratadores das exceções e que separe estes tratadores da atividade normal da aplicação. Em adição, esta infra-estrutura deveria promover a separação entre os componentes contendo os tratadores de exceções e os componentes responsáveis pela invocação do tratador apropriado. As seguintes forças estão associadas a este problema de projeto:

- tratadores de exceções para exceções locais e concorrentes deveriam ser definidas de forma uniforme; e
- a arquitetura de software deveria incluir associação de tratadores em multi-níveis. Isto é, a associação de tratadores a regiões protegidas de diversos níveis tais como classes, objetos, métodos, etc.

Solução

Nós utilizamos a técnica de reflexão computacional (Seção 4.3) com o objetivo de separar as classes responsáveis pela invocação dos tratadores (meta-nível) das classes usadas para especificar os tratadores (nível base). O nível base define as classes excepcionais, isto é, classes da aplicação que implementam os tratadores para as exceções locais e concorrentes. Os métodos destas classes são os tratadores das exceções levantadas durante a execução dos métodos das classes normais. As classes normais estão presentes no nível base e implementam as atividades normais da aplicação. Conseqüentemente, as classes excepcionais estão associadas a classes normais. O meta-nível consiste de meta-objetos que estão associados a classes excepcionais, e são responsáveis pela invocação transparente dos tratadores.

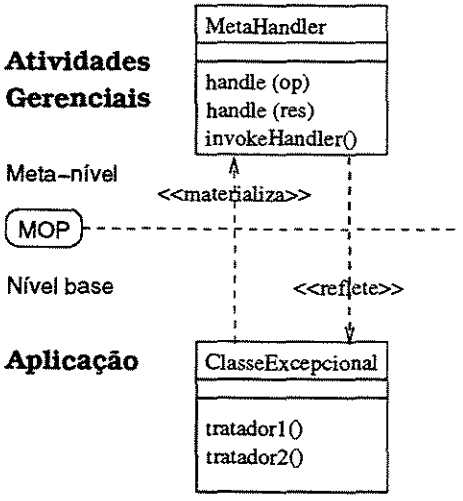


Figura 5.5: Padrão tratador

Estrutura

O padrão **tratador** [51, 53] consiste de dois tipos de elementos: (i) classes excepcionais e (ii) meta-objetos, instâncias da meta-classe MetaHandler. As classes excepcionais

estão presentes no nível base e definem as atividades de tratamento de erros de uma aplicação específica. Os métodos das classes excepcionais são os tratadores das exceções locais e concorrentes levantadas durante a execução dos métodos das classes normais. Meta-objetos, instâncias de MetaHandler, estão associados a classes excepcionais e são responsáveis pela invocação transparente dos tratadores de exceções. Classes excepcionais contém tratadores de exceções (locais e/ou concorrentes) associados a classes, objetos e métodos (associação de tratadores em multi-níveis).

Dinâmica

O seguinte diagrama (Figura 5.6) ilustra um cenário típico deste padrão. Neste cenário, suponha que um método da aplicação levante a exceção Exception e o tratador associado a esta exceção foi executado com sucesso.

1. O meta-objeto, instância de MetaSearcher (Seção 5.1.2), associado ao objeto da aplicação que levantou a exceção, intercepta a ocorrência da exceção e solicita ao meta-objeto (instância MetaHandler) associado ao tratador desta exceção que inicie o seu tratamento.

2. O MetaHandler invoca transparentemente o método de ClasseExcepcional que implementa o tratador para esta exceção.

3. O tratador de exceção então invoca o método getLocation de Exception (Seção 5.2.1) com o objetivo de acessar a informação relacionada a localização da ocorrência da exceção.

4. O tratador de exceção então retorna o resultado do tratamento da exceção para o MetaHandler que o devolve para o MetaSearcher.

Variantes

- **Variante totalmente reflexiva.** Esta variante transfere os métodos que implementam os tratadores das classes excepcionais para o meta-nível. Isto é, a classe excepcional e a meta-classe MetaHandler são unidas e tornam-se apenas uma meta-classe. Os meta-objetos associados a instâncias de classes normais contem métodos que são responsáveis por realizar o tratamento de exceções. Ao invés de usar a técnica de reflexão computacional para alcançar a completa separação entre a aplicação e mecanismos de gerenciamento, esta variante explora reflexão computacional para separar os códigos normais e excepcionais de uma aplicação.

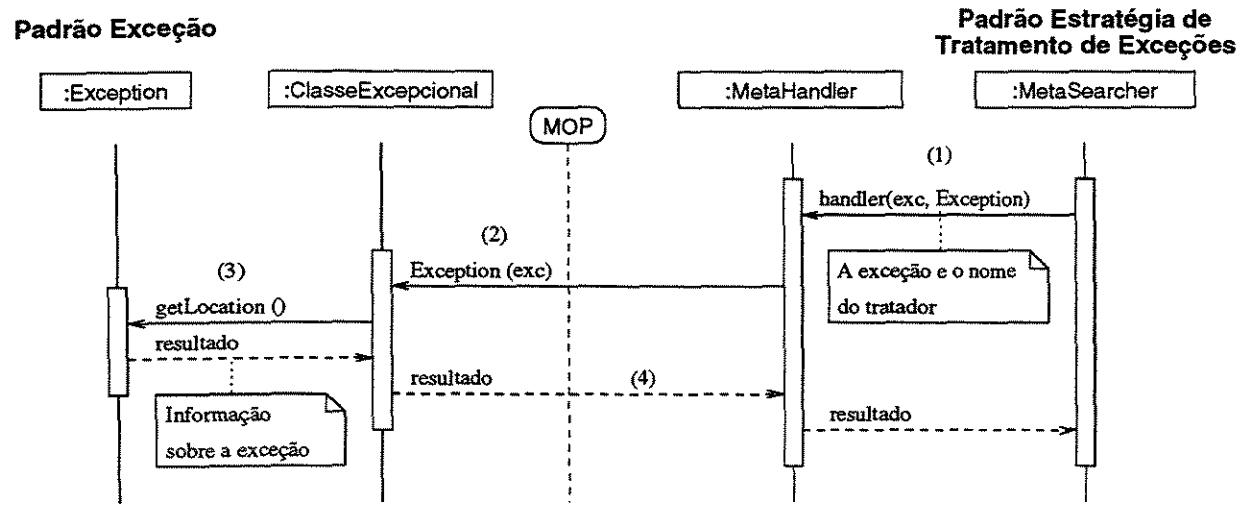


Figura 5.6: Dinâmica: padrão tratador

Usos conhecidos

- Hof *et al.* [62] também usa a técnica de reflexão computacional objetivando obter meta-informação sobre a aplicação e invocar o tratador apropriado quando uma exceção é levantada. Estruturas presentes no meta-nível implementam os mecanismos de tratamento de exceções enquanto no nível base reside a aplicação. Finalmente, tratadores também são implementados como métodos.

- Mitchell *et al.* [90] usa a variante totalmente reflexiva discutida anteriormente.

Conseqüências

- **Uniformidade.** Tratadores para as exceções locais e concorrentes são definidas uniformemente como métodos em classes excepcionais.

- **Legibilidade.** Este padrão provê separação explícita entre as atividades normais e as atividades de tratamento de erros, que conseqüentemente promove legibilidade e facilidade de manutenção.

- **Flexibilidade.** A associação de tratadores em multi-níveis permite projetistas associar tratadores a classes, objetos e métodos.

- **Reutilização.** O uso de classes normais e excepcionais permitem projetistas comporem uma hierarquia de classes excepcionais que é ortogonal à hierarquia de classes

normais da aplicação. A hierarquia de classes excepcionais permite subclasses excepcionais herdar tratadores de suas superclasses e, conseqüentemente, possibilita a reutilização de código responsável pelo tratamento de erros. Quando a reutilização não é desejada, os tratadores podem ser redefinidos nas subclasses.

- **Ausência de checagem.** Uma possível desvantagem deste padrão é que não é fácil checar se foram definidos tratadores para todas as exceções da aplicação.

5.1.4 Padrão redundância de software

Contexto

Falhas de software são sempre falhas de projeto. Para construir software que trata falhas de seus componentes, nós necessitamos de técnicas que lidam com falhas de projeto. Técnicas para obtenção de tolerância a falhas em software dependem do emprego efetivo de redundância [74].

Problema

A incorporação de redundância em um sistema deve ser realizada de forma estruturada e disciplinada, caso contrário pode aumentar a complexidade e o custo do desenvolvimento do sistema.

Solução

Nós utilizamos um padrão geral para o domínio de tolerância a falhas, que provê uma solução uniforme para a incorporação de redundância em um sistema orientado a objetos. Este padrão, chamado **redundância de software** [14, 45, 46] apresenta a mesma estrutura do padrão **state reflexivo** [46] mas com semânticas diferentes.

Estrutura

As classes do nível base representam o componente tolerante a falhas (ComponenteTF) que define os serviços tolerantes a falhas e os componentes redundantes (subclasses de ComponenteRed) que implementam diferentes versões para os serviços providos por ComponentTF. O ComponenteTF deve implementar uma entre duas interfaces: RBInterface ou NVInterface. Estas interfaces provém um método (accTest) que será invocado para verificar se a operação (service) foi executada com sucesso. Estas interfaces implementam o

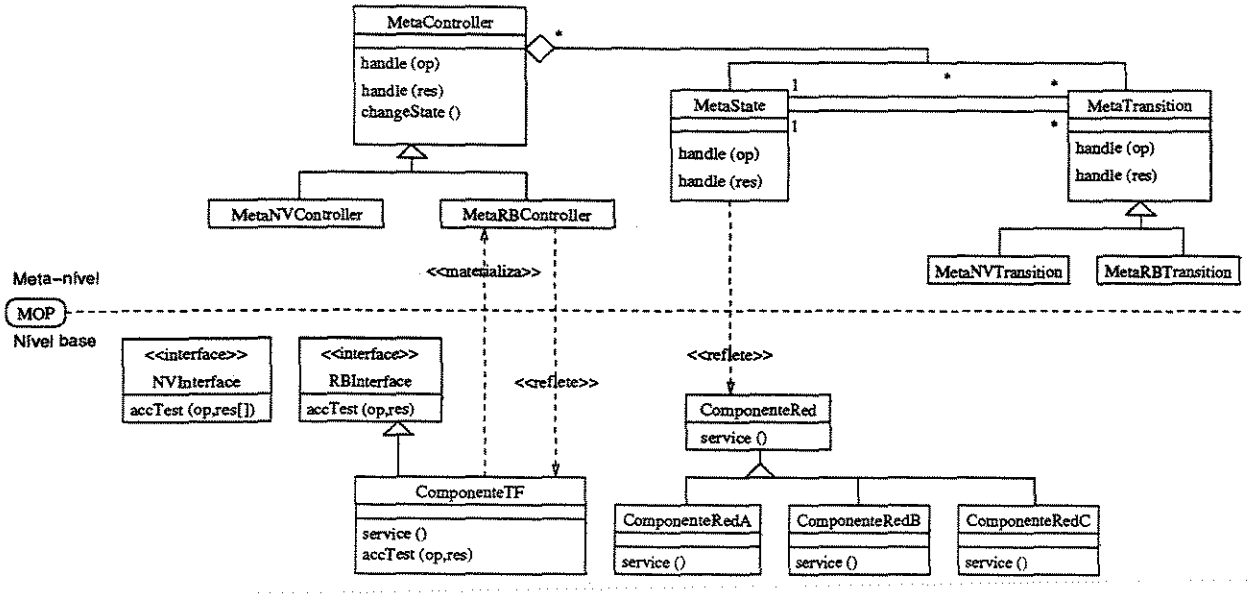


Figura 5.7: Padrão redundância de software

teste de aceitação da técnica de bloco de recuperação [97] (Seção 2.4.1) ou o árbitro da técnica de N-versões [4] (Seção 2.4.2). Este método (accTest) retorna true se a operação foi executada com sucesso, e false caso contrário. As classes no meta-nível implementam os mecanismos correspondentes a cada técnica. Por exemplo, instâncias de MetaState ativam os diferentes componentes redundantes e instâncias das subclasses de MetaTransition coletam os resultados (ou o resultado único, em caso de blocos de recuperação) da invocação dos serviços redundantes e invocam o método accTest que implementa o teste de aceitação ou o árbitro como discutido anteriormente. Na Figura 5.7, a técnica utilizada é o bloco de recuperação e portanto, instâncias de ComponenteTF implementarão a interface RBInterface e serão associadas a instâncias de MetaRBController. O salvamento e a recuperação dos estados de computação de ComponenteTF é feita transparentemente pela meta-configuração associada a este componente.

Dinâmica

Os seguintes diagramas ilustram dois cenários típicos deste padrão:

Cenário I. O primeiro diagrama (Figura 5.8) ilustra um cenário típico do padrão sendo utilizado para implementar a técnica de bloco de recuperação. Neste exemplo, a primeira versão do serviço foi executado sem sucesso, então a segunda é executada com sucesso, retornando o resultado desejado.

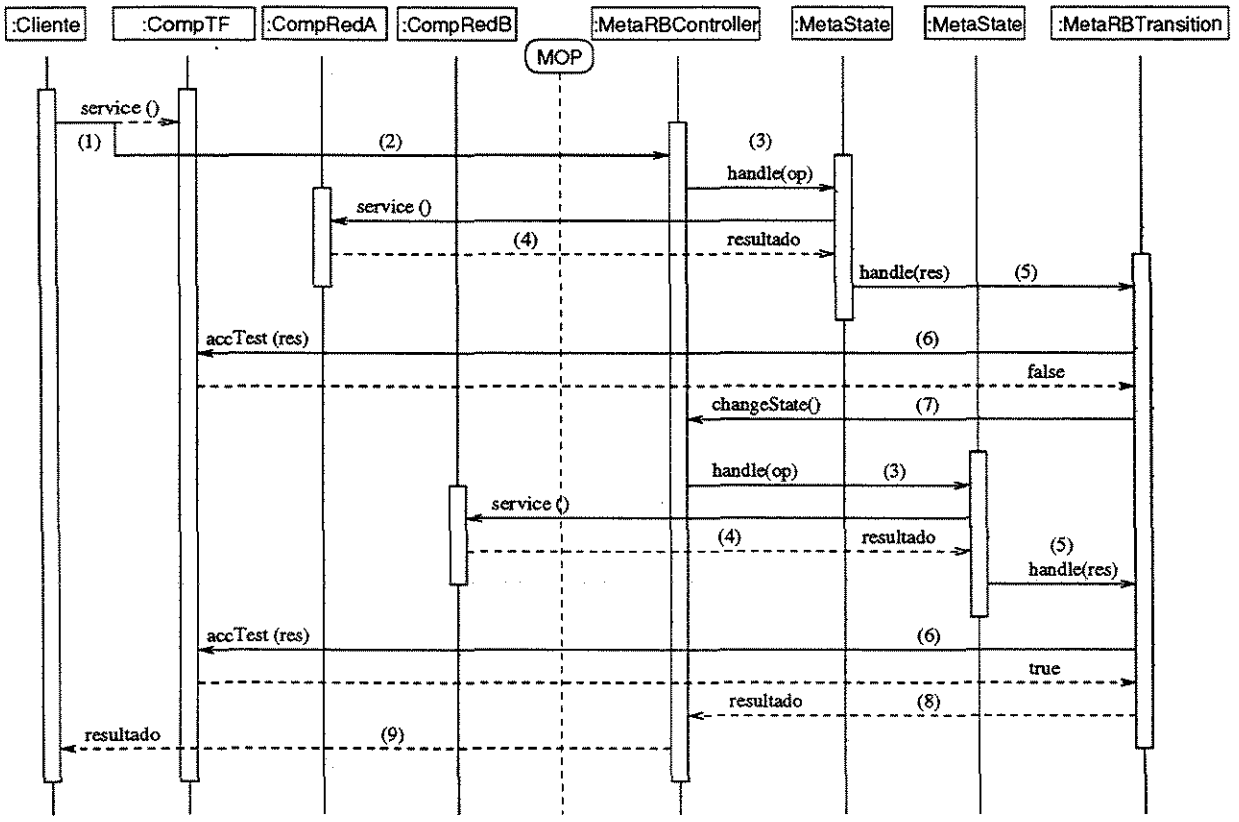


Figura 5.8: Dinâmica: padrão redundância de software (bloco de recuperação)

1. O Cliente solicita um serviço para ComponenteTF;

2. O MetaRBController intercepta esta invocação de operação;

3. O MetaRBController delega esta requisição de serviço para o MetaState corrente. Se cada versão alternativa já foi executada sem sucesso, então a exceção FailedServiceException é levantada;

4. O MetaState corrente invoca a operação na versão alternativa associada a ele e intercepta o resultado desta operação;

5. O MetaState então delega este resultado para o MetaRBTransition;

6. O MetaRBTransition analisa o resultado (invoca a operação accTest em ComponenteTF);

7. Se o resultado não passou pelo teste de aceitação, então o MetaRBTransition solicita ao MetaRBController que mude o MetaState corrente (volte para o passo 3);

8. Se o resultado passou pelo teste de aceitação, ele retorna o resultado para o MetaRBController;

9. O MetaRBController então devolve o resultado para o Cliente.

Cenário II. O segundo diagrama (Figura 5.9) ilustra um cenário típico do padrão sendo utilizado para implementar a técnica N-Versões.

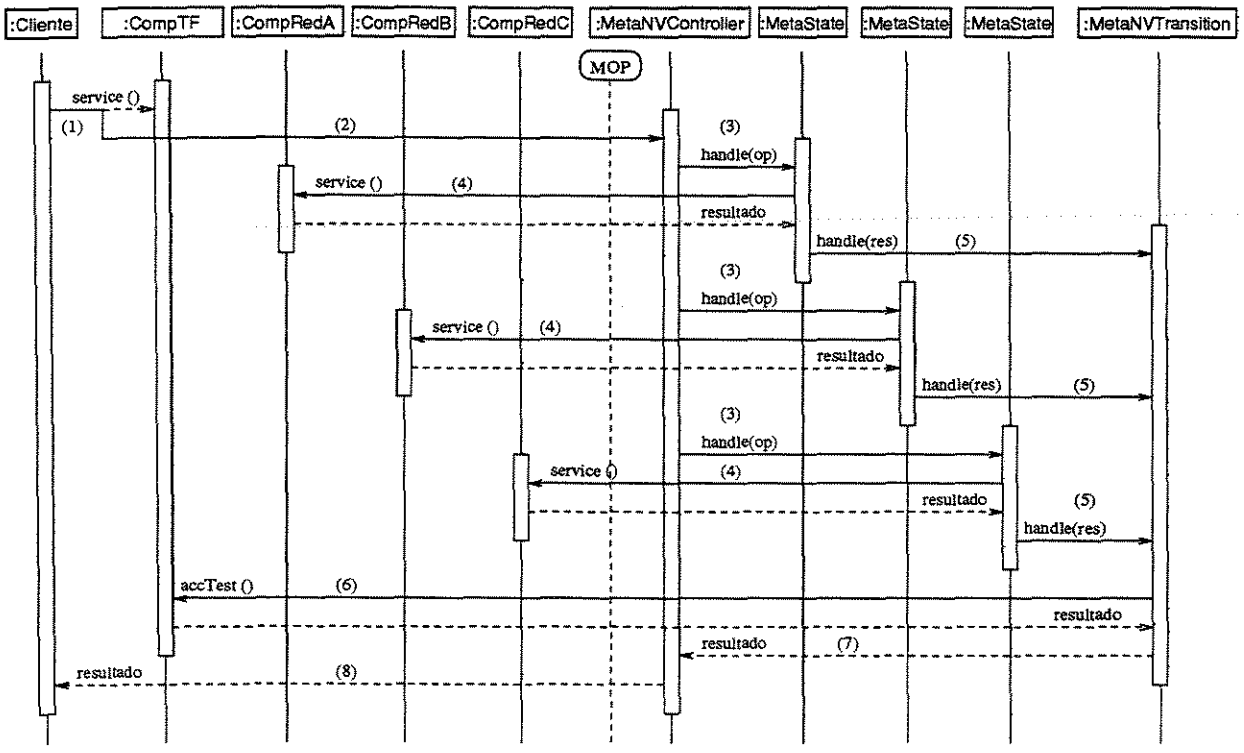


Figura 5.9: Dinâmica: padrão redundância de software (n-versões)

1. O Cliente solicita um serviço para ComponenteTF;

2. O MetaNVController intercepta esta invocação de operação;

3. O MetaNVController delega esta requisição de serviço para o conjunto de MetaStates.

4. Os MetaStates invocam a operação na versões associadas a eles e interceptam os resultados desta operação;

5. Os MetaStates então delegam os resultado para o MetaNVTransition;

6. O MetaNVTransition analisa os resultados (invoca a operação accTest em ComponenteTF);

7. Se o serviço foi executado corretamente, então o MetaNVTransition então devolve o resultado (por exemplo, a maioria) para o MetaNVController. Caso contrário, a exceção FailedServiceException é levantada;

8. O MetaNVController então devolve o resultado para o Cliente.

Variantes

- **Variante não reflexiva.** Esta variante transfere todas as atividades gerenciais (invocação dos projetos redundantes, teste de aceitação, salvamento e restauração do estado do componente) para o nível base.

Usos conhecidos

- Tolerância a falhas de software têm sido utilizados em um grande número de aplicações. Na maioria dos casos, em aplicações críticas tais como plantas nucleares e sistemas aeroespaciais [82, 81].

- Nós identificamos pelo menos três padrões de projeto que usam a variante não reflexiva discutida anteriormente: os padrões **master-slave** [24] e **backup** [120] apresentam soluções de projeto para a implementação das técnicas de N-Versões [4] e Bloco de Recuperação [97] respectivamente. O padrão **reliable hybrid** [33] permite reusar este dois padrões, bem como construir soluções híbridas mais complexas. O padrão **reliable hybrid** é baseado na utilização do padrão **Composite** [49] para recursivamente combinar os padrões **master-slave** e **backup**.

Conseqüências

- **Aumento da confiabilidade do sistema.** Estudos mostram que dependendo do tipo de sistema, o uso de redundância proporciona um aumento na confiabilidade do sistema no mínimo de 30% [81]. Uma análise dos custos e de quão críticos são os componentes deveria ser feita para decidir se os ganhos são suficientes e recompensadores.

- **Custo.** É óbvio que o desenvolvimento de diversas versões da mesma funcionalidade, custará mais do que o desenvolvimento de apenas uma versão. Experiência prática

com o desenvolvimento de software usando componentes redundantes indica que, em um sistema bem projetado, cada componente adicional custa cerca de 75-80% [82] do custo de uma única versão.

- **Transparência.** O salvamento e recuperação dos estados dos componentes redundantes é realizado transparentemente pela meta-configuração associada ao componente e projetistas da aplicação não necessitam se preocuparem em implementar métodos que implementam tais tarefas.

5.1.5 Projeto de um componente tolerante a falhas ideal

Os padrões discutidos nas seções anteriores apresentam soluções de projeto para a implementação de componentes tolerantes a falhas ideais. Como discutido anteriormente, meta-objetos são responsáveis pela implementação de propriedades não-funcionais de uma aplicação.

Para obter uma estruturada composição de propriedades não-funcionais, definimos um tipo especial de meta-objeto chamado delegador (Seção 4.3) que delega operações e resultados para outros meta-objetos. O padrão de projeto **cadeia de responsabilidades** [49] provê o projeto detalhado desta composição. Este padrão possibilita que mais do que um objeto trate uma requisição de serviço por construir uma cadeia de objetos e passar a requisição através desta cadeia até que um ou mais objetos tratem esta requisição.

Figura 5.10 ilustra um diagrama de interação entre os padrões **contrato reflexivo**, **estratégia de tratamento de exceções**, **tratador** e **redundância de software**. Nesta figura, Componente está associado a um meta-objeto (Delegador) que delega operações e resultados para outros 3 meta-objetos.

1. Enquanto executando a sua tarefa, Cliente envia uma requisição de serviço ao Componente. Delegador intercepta esta requisição de serviço e delega esta para MetaContract.

2. MetaContract recebe esta requisição de operação e chama o teste de precondição e devolve uma exceção se o teste não é satisfeito. Se uma exceção for devolvida, Delegador já delega esta exceção como resultado da operação para MetaSearcher (passo 5), não delegando a operação para o próximo meta-objeto. Se o teste de precondição é satisfeito, ele devolve noResult que informa ao Delegador que ele pode delegar esta operação para o próximo meta-objeto.

3. O próximo meta-objeto é opcional. Um componente tolerante a falhas ideal pode ser definido sem usar redundância. Então existe duas possibilidades: (a) Caso a

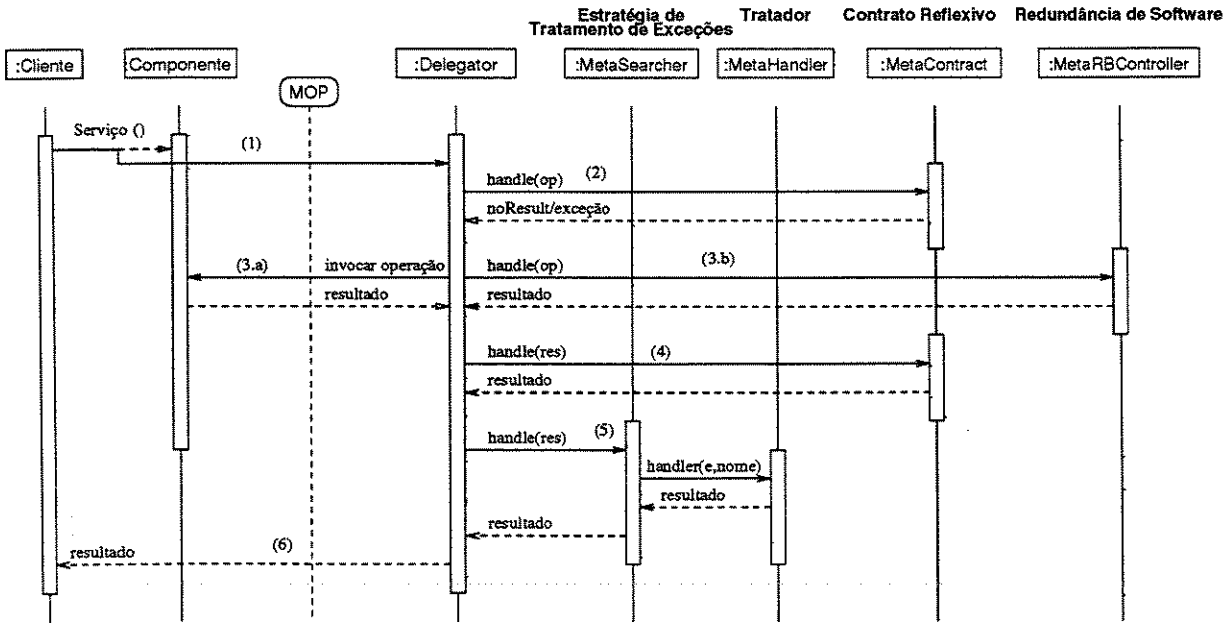


Figura 5.10: Dinâmica: componente tolerante a falhas ideal

redundância não seja utilizada, então a operação é invocada em Componente presente no nível base e o resultado desta operação é devolvido para Delegador; (b) Caso contrário, Delegador então delega a operação para MetaRBController (ou MetaNVController) que invoca as variantes selecionando e devolvendo o resultado da execução destas ou uma exceção caso todas as variantes falharem.

4. Delegador recebe o resultado e delega para MetaContract que chama o teste de pós-condição devolvendo uma exceção se este teste não é satisfeito. Caso o teste seja satisfeito ou o resultado seja uma exceção, então devolve o mesmo resultado para Delegador.

5. Delegador recebe o resultado devolvido e delega para MetaSearcher. Se o resultado não é uma exceção, devolve o mesmo resultado para Delegador. Caso contrário, chama o tratador associado a esta exceção e devolve o resultado da execução do tratador. Se o tratador não for encontrado, devolve uma exceção de defeito como resultado da operação.

6. Delegador recebe o resultado e devolve este para Cliente como resultado da operação requisitada no componente. Note que desde que a definição de componente tolerante a falhas ideal é recursiva, este resultado poderia ser testado (contratos) ou tratado (tratamento de exceções) caso Cliente estivesse associado a uma meta-configuração semelhante.

5.2 Respostas excepcionais

Nesta seção apresentamos um padrão que provê uma solução de projeto para a implementação dos conectores do estilo arquitetural componente tolerante a falhas ideal (Seção 4.1).

5.2.1 Padrão exceção

Contexto

Projetistas de sistemas confiáveis deveriam ser aptos a especificar as exceções locais e concorrentes de suas aplicações. Estas exceções podem ser levantadas durante a execução do sistema. Extra-informações que possibilitem o tratamento dessas ocorrências excepcionais são requeridas pelas aplicações.

Problema

A arquitetura de software deveria dar suporte a definição de exceções locais e concorrentes. Além disso, uma arquitetura de software flexível e reutilizável é requerida com o objetivo de facilitar a especificação das exceções e apresentar uma separação entre as exceções da aplicação e o gerenciamento das extra-informações necessárias ao tratamento destas. Diversas forças estão associadas a este problema de projeto:

- Exceções locais e concorrentes deveriam ser definidas de forma uniforme;
- Projetistas de software deveriam ser aptos a construir árvores de exceção [27] facilmente; e
- A exceção possui extra-informação necessária para o seu tratamento.

Solução

Nós utilizamos a técnica de reflexão computacional (Seção 4.3) com o objetivo de separar as classes responsáveis pelo gerenciamento das extra-informações (meta-nível) daquelas usadas para especificar as exceções (nível base). Diferentes tipos de exceções são organizadas hierarquicamente como classes que derivam da classe raiz Exception. Árvores de exceções são definidas através do uso do padrão de projeto **composite** [49]. Exceções são objetos do nível base criados durante o tempo de execução do sistema quando algum erro é detectado. Meta-objetos são associados aos objetos que representam as exceções e

mantém extra-informações sobre ocorrências de exceções. Meta-objetos alteram transparentemente o estados destes objetos com o objetivo de deixar estas informações disponíveis para a aplicação. Como resultado, as exceções mantém extra-informações necessárias ao seu próprio tratamento. A aplicação obtém esta informação por invocar métodos nos objetos que representam as exceções.

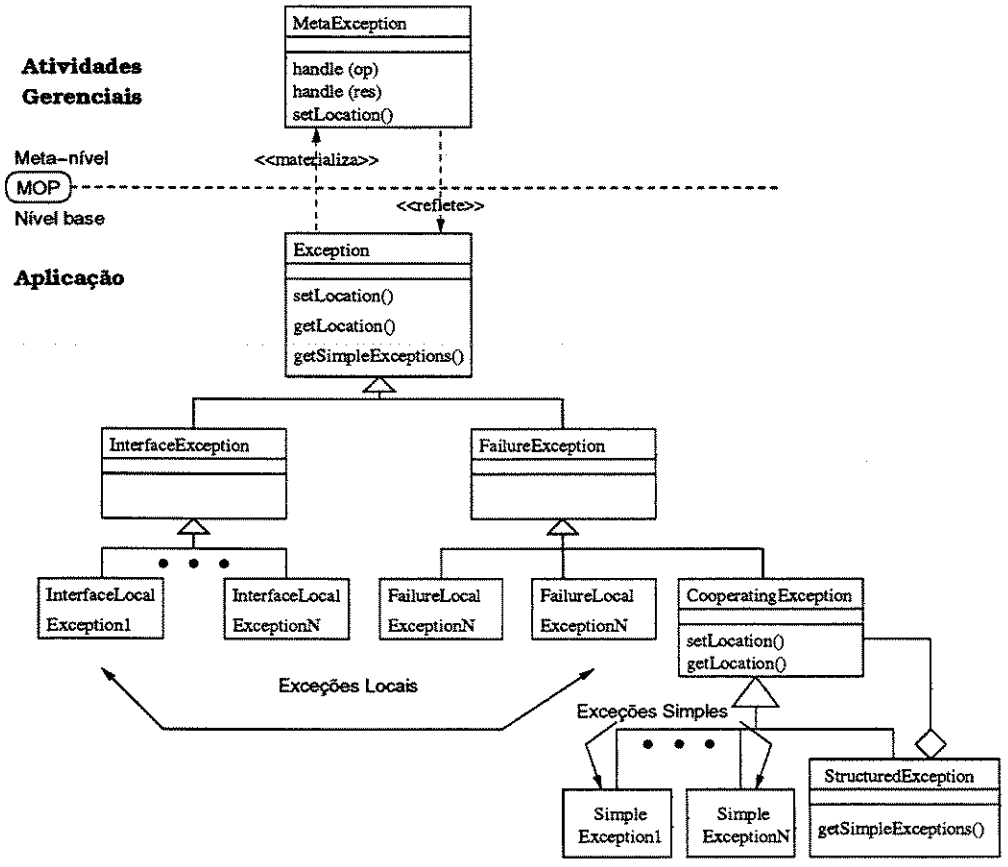


Figura 5.11: Padrão exceção

Estrutura

O padrão **exceção** [51, 53] consiste de classes que representam as exceções e a meta-classe MetaException. Instâncias de MetaException estão associadas à instâncias das subclasses de Exception presentes no nível base. Este padrão define quatro tipos de exceções - InterfaceException, FailureException, CooperatingException e StructuredException (Figura 5.11) - que são disponibilizadas para serem usadas por projetistas de aplicações. As classes InterfaceException e FailureException define as exceções de interface e de defeito das aplicações; ela é derivada por projetistas de aplicações no intuito de representar as exceções

locais de uma aplicação específica. Em nossa abordagem, as exceções concorrentes são exceções de defeito. Árvores de exceções são facilmente especificadas - um projetista apenas necessita criar uma classe para cada exceção simples (folhas da árvore) por derivar a classe CooperatingException, e uma nova instância da classe StructuredException para cada exceção estruturada de sua aplicação. Uma instância de StructuredException armazena as exceções simples e/ou estruturadas que o compõe. O método getSimpleException retorna as exceções simples que compõe uma exceção estruturada. Este esquema para a definição de árvores de exceção é similar à estrutura definida pelo padrão de projeto **composite** [49].

Dinâmica

O seguinte diagrama (Figura 5.12) ilustra um cenário típico deste padrão. Neste cenário, uma instância de Exception foi levantada e a informação sobre a localização da ocorrência da exceção é atualizada:

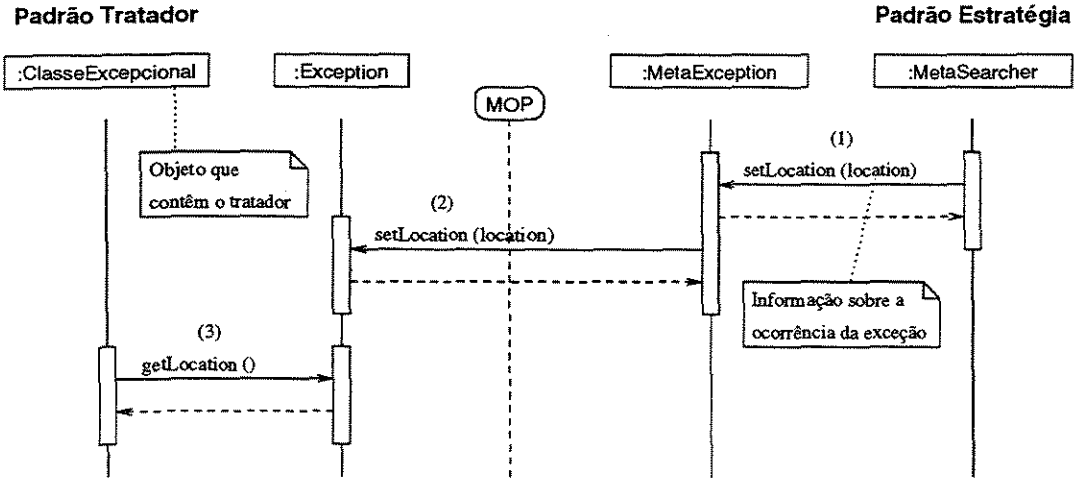


Figura 5.12: Dinâmica: padrão exceção

1. O meta-objeto, instância de MetaSearcher (Seção 5.1.2), associado ao objeto da aplicação que levantou a exceção, materializa a informação sobre a localização da ocorrência da exceção. Este meta-objeto envia esta informação para o meta-objeto (instância de MetaException) associado a Exception.

2. MetaException então atualiza a informação sobre a ocorrência da exceção por invocar transparentemente o método setLocation em Exception. A invocação e atualização é transparente sobre o ponto de vista da aplicação.

3. O tratador de exceção (Seção 5.1.3) então invoca o método getLocation com o objetivo de acessar a informação relacionada a localização da ocorrência da exceção.

Usos conhecidos

- A representação de exceções como objetos é uma solução de projeto adotadas em diversos sistemas e linguagens de programação, tais como Java [48], C++ [119] e Arche [16].

Conseqüências

- **Uniformidade.** Ambas as exceções locais e concorrentes são uniformemente definidas como instâncias da classe Exception. Além disso, este padrão permite projetistas de aplicações tratar exceções simples e suas composições (exceções estruturadas) uniformemente desde que ele adota a estrutura do padrão de projeto **composite** [49].
- **Simplicidade.** Árvores de exceções são facilmente definidas. Projetistas de aplicações definem as árvores de exceções sem precisar escrever um procedimento de resolução de exceções para cada ação atômica de sua aplicação. Em adição, o procedimento de resolução de exceções é realizado transparentemente pelo meta-nível (Seção 5.4.1).
- **Reutilização.** A representação de exceções locais e concorrentes como objetos promove a reutilização das classes que representam as exceções.
- **Legibilidade.** Aplicações onde as exceções são representadas como objetos são mais fáceis de entender e manter do que aquelas em que exceções são meros símbolos (números ou cadeias de caracteres).
- **Fácil incorporação de tratadores genéricos.** Desde que exceções são representadas como objetos, tratadores genéricos podem ser definidos como métodos das classes que representam as exceções. No caso em que projetistas não tenham definidos tratadores mais específicos, este método poderia ser invocado pelo meta-nível.

5.3 Papéis de componentes

Nesta seção apresentamos um padrão que provê uma solução de projeto para a implementação dos componentes lógicos do estilo arquitetural colaborações entre papéis (Seção 4.2).

5.3.1 Padrão papel reflexivo

Contexto

Um sistema orientado a objetos é tipicamente baseado em um conjunto de entidades. Cada entidade é modelada por uma classe correspondente em termos de seu estado e comportamento abstrato. Esta abordagem geralmente funciona no projeto de pequenas aplicações. Entretanto, quando desejamos construir sistemas mais complexos, nós temos que lidar com diferentes clientes que necessitam de diferentes visões de nossas entidades. O exemplo mais ilustrativo é a entidade pessoa. Uma pessoa pode ser vista como um empregado de uma empresa, um professor universitário, etc.

Problema

O desafio é representar todas estas visões consistentemente de tal forma que se uma visão for modificada, os clientes que não estão interessados naquela visão não sejam afetados pela mudança.

Solução

Papéis [71, 128] representam as diferentes visões de uma entidade (objeto). As características inerentes ao conceito de papéis são os seguintes: (i) uma entidade pode desempenhar diferentes papéis em diferentes contextos; (ii) o conjunto de papéis que uma entidade pode desempenhar varia com o tempo, então seu comportamento varia também; (iii) os papéis de uma entidade compartilham estado e comportamento. Por exemplo, os papéis estudante e empregado da entidade pessoa compartilham o seu nome e o número de seu telefone residencial e (iv) entidades exibem comportamento específicos ao papel em que ele está desempenhando. Por exemplo, o salário de empregado não é acessível se pessoa está desempenhando o papel de estudante.

Nós propomos o padrão de projeto chamado **papel reflexivo** [14] que captura o conceito de papéis. Neste padrão existe a separação entre a entidade e a hierarquia de papéis que ele pode desempenhar. Este padrão associa uma entidade com diferentes "objetos papéis", cada um representando um papel que a entidade desempenha nos diferentes contextos sendo a gerência dos papéis feita no meta-nível. Por representar papéis como objetos individuais, diferentes contextos são mantidos separados e portanto a configuração do sistema é simplificada.

Estrutura

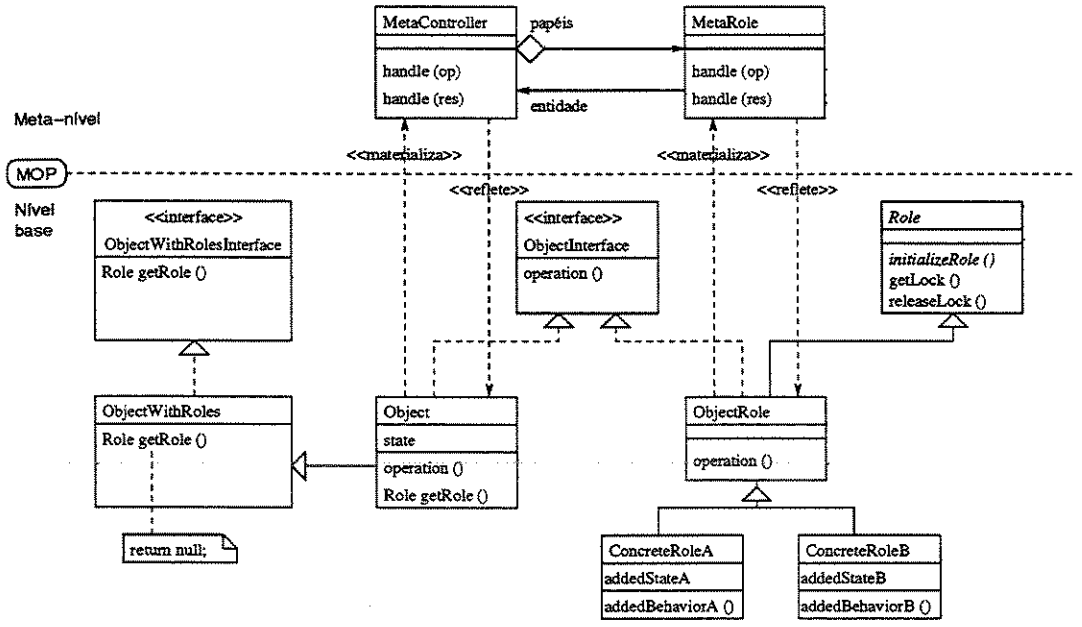


Figura 5.13: Padrão papel reflexivo

Nós utilizamos a técnica de reflexão computacional com o objetivo de separar os requisitos funcionais dos requisitos gerenciais da aplicação. Além disso, existe a separação entre a entidade e a hierarquia de papéis que ele pode desempenhar. As classes do nível base representam a entidade (Object) e as diferentes visões (papéis) desta entidade (Figura 5.13). A classe raiz da hierarquia de papéis (ObjectRole) deve ser subclasse da classe Role e implementar a mesma interface (ObjectInterface) que a entidade implementa. Desta forma, cada papel implementa pelo menos a mesma interface da entidade adicionando novas operações. Se a entidade Object não possui superclasses, então ela pode estender a classe ObjectWithRoles que implementa a interface ObjectWithRolesInterface. Caso contrário, ela deve implementar a interface ObjectWithRolesInterface. Neste caso, ela deve incluir a implementação vazia do método getRole (como feita pela classe ObjectWithRoles na Figura 5.13).

O meta-nível define dois tipos de objetos: instâncias de MetaController e MetaRole. Instâncias de MetaController estão associadas a entidades (Object) e são responsáveis, entre outras atividades, pela criação transparente dos papéis e pela associação destes à instâncias de MetaRole. A cada operação requisitada em um papel, o meta-objeto associado (instância de MetaRole) verifica se é uma operação específica para aquele papel (addedBehaviorA presente na interface de ConcreteRoleA) ou uma operação compartilhada pelos diversos papéis (operation presente na interface de Object). No primeiro caso, a

operação é executada pelo próprio papel. No segundo caso, a operação é delegada para a entidade que executa a operação e retorna para o cliente da operação. A delegação é implementada no meta-nível.

Manter a consistência entre papéis torna-se difícil desde que uma entidade desempenha diversos papéis que são mutuamente independentes. A abordagem proposta por este padrão baseia-se em um mecanismo de controle de concorrência em que atributos da entidade são bloqueados/desbloqueados com o objetivo de preservar tal consistência. Subclasses de Role devem redefinir o método getLock que especifica quais os atributos da entidade que devem ser bloqueados por este papel. Caso este método não seja redefinido, é assumido que **todos** os atributos da entidade serão bloqueados.

Dinâmica

Os seguintes diagramas ilustram dois cenários típicos deste padrão:

Cenário I. O primeiro cenário (Figura 5.14) ilustra a invocação de uma operação compartilhada pelos papéis. Neste caso, a operação é delegada para o objeto (Object) que executa a operação e retorna o seu resultado.

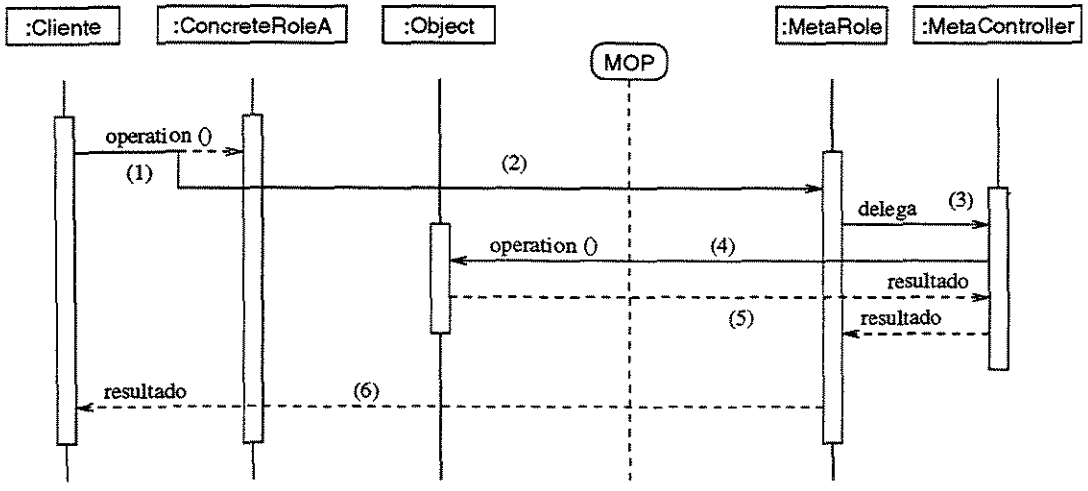


Figura 5.14: Cenário I: invocação de uma operação compartilhada

1. O Cliente invoca operation (uma operação compartilhada pelos papéis) em ConcreteRoleA;

2. O MetaRole intercepta esta invocação de operação e verifica se ela é uma operação específica ao papel associado a este meta-objeto;

3. Se a operação invocada não é uma operação específica ao papel, então o MetaRole delega esta operação para o MetaController;

4. O MetaController invoca a operação em Object;

5. O MetaController intercepta o resultado desta invocação e retorna o resultado para MetaRole;

6. Ao receber o resultado da operação delegada, o MetaRole retorna este resultado para o Cliente.

Cenário II. O segundo cenário (Figura 5.15) ilustra a invocação que é específica de um papel. Neste caso, a operação é executada pelo próprio papel.

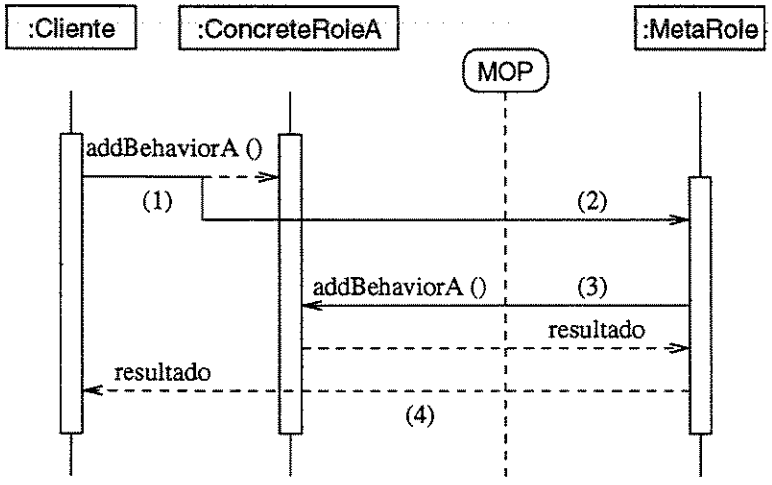


Figura 5.15: Cenário II: invocação de uma operação específica a um papel

1. O Cliente invoca a addBehaviorA que é uma operação específica ao papel ConcreteRoleA;

2. O MetaRole intercepta esta invocação de operação e verifica se ela é uma operação específica ao papel associado a este meta-objeto;

3. Se a operação invocada é uma operação específica ao papel, então o MetaRole invoca esta operação em ConcreteRoleA;

4. O MetaRole intercepta o resultado desta invocação e retorna o resultado para o Cliente.

Variantes

- **Não reflexiva.** Esta variante foi proposta por Bäumer *et al.* [7]. Nesta variante, a gerência relacionada à criação dos papéis e a delegação de operações (papéis para a entidade) deve ser implementada pelos programadores da aplicação no nível base. Conseqüentemente não existe a separação entre os requisitos funcionais e gerenciais da aplicação.

Usos conhecidos

- Em pesquisas na área de projetos baseados em colaborações[1] [125, 128], papéis prescrevem a atividade de um objeto no contexto de uma colaboração.

- Em pesquisas na área de banco de dados orientado a objetos [1, 57], papéis são utilizados para modelar o comportamento de entidades do mundo real que evoluem durante o tempo de vida do sistema.

Conseqüências

- **Legibilidade.** A definição da entidade pode ser feita de forma concisa. A interface da entidade é bem focada nas suas características essenciais e não é "poluída" com características específicas dos papéis que esta entidade desempenha.

- **Consistência.** Manter a consistência entre papéis torna-se difícil desde que uma entidade desempenha diversos papéis que são mutuamente independentes. A abordagem proposta por este padrão baseia-se em um mecanismo de controle de concorrência em que atributos da entidade são bloqueados/desbloqueados com o objetivo de preservar tal consistência. Conseqüentemente, a responsabilidade da correta especificação dos atributos a serem bloqueados fica a cargo dos projetistas da aplicação.

- **Separação de interesses.** Por explicitamente separar a entidade dos papéis que este desempenha, a dependência de aplicações que utilizam diferentes papéis desta entidade é minimizada. Uma aplicação (clientA) usando um papel da entidade não necessita ter conhecimento dos outros papéis utilizados por outras aplicações (clientB).

[1] Em inglês: *collaboration-based design.*

5.4 Conector: colaborações

Nesta seção apresentamos um padrão que provê uma solução de projeto para a implementação dos conectores do estilo arquitetural colaborações entre papéis.

5.4.1 Padrão colaboração confiável

Contexto

Projetistas de aplicações confiáveis deveriam ser aptos a especificar atividades concorrentes cooperativas. Estas atividades deveriam ser controladas em tempo de execução e seus participantes deveriam deixar a atividade de forma síncrona. Durante a execução da atividade cooperativa, exceções podem ser levantadas. Como conseqüência, um processo de resolução de exceções concorrentes é necessário para determinar a exceção resolvida que deve ser tratada por todos os participantes da atividade cooperativa. Entretanto, se não for possível tratar esta exceção (isto é, a recuperação de erros por avanço não teve sucesso), a recuperação de erros por retrocesso deveria ser aplicada objetivando restaurar o sistema ao estado anterior à ocorrência do erro. E por fim, a consistência de recursos externos acessados por diferentes atividades cooperativas deveria ser mantida.

Problema

A arquitetura de software deveria dar suporte à definição de atividades concorrentes cooperativas. Além disso, uma abordagem disciplinada é requerida objetivando separar interesses e minimizar dependências entre as atividades concorrentes cooperativas da aplicação e a estratégia para a recuperação de erros por avanço (resolução de exceções) e por retrocesso (restauração dos estados computacionais). Algumas forças estão associadas a este problema de projeto:

- A definição das atividades cooperativas deveria ser feita de forma estruturada a fim de evitar um aumento na complexidade do software;
- A estratégia para tratamento de exceções concorrente deveria ser uma extensão consistente da estratégia geral para tratamento de exceções (Seção 5.1.2);
- A abordagem bloqueadora deveria ser usada para tratamento de exceções concorrentes desde que ele é mais simples e fácil de implementar (Seção 2.5.3).

- A recuperação de erros por retrocesso deveria ser feita de forma transparente para os programadores da atividades concorrentes.

Solução

Nós utilizamos a técnica de reflexão computacional para separar as classes responsáveis pelos mecanismos gerenciais (meta-nível) das classes que definem as atividades cooperativas da aplicação (nível base). O padrão **colaboração confiável** [15, 11] separa os objetos em dois bem definidos níveis (Figura 5.16) e provê uma solução de projeto para a implementação da técnica de ações atômicas coordenadas (Seção 2.5.3). O nível base provê aos projetistas de aplicações, classes que definem as atividades cooperativas da aplicação; a definição de atividades aninhadas é também permitida com o objetivo de controlar a complexidade do sistema e permitir uma melhor organização das atividades normais e de tratamento de erros do sistema. O protocolo de meta-objetos intercepta e materializa invocações de métodos e resultados. O meta-nível implementa os mecanismos gerenciais baseados nas invocações de métodos e resultados materializados e na meta-informação disponível. Desta forma, meta-objetos são responsáveis, entre outras atividades, pela sincronização dos participantes da atividade cooperativa, pela resolução de exceções, pela invocação do tratador da exceção e caso a recuperação de erros por avanço não tenha sucesso, realizar a recuperação de erros por retrocesso.

Estrutura

Este padrão introduz cinco classes: Collaboration, Participant, MetaParticipant, MetaCollaboration e MetaAtomic (Figura 5.16). Projetistas estendem as classes Collaboration e Participant adicionando funcionalidades dependentes de aplicação. Em nossa abordagem, instâncias das subclasses de Collaboration correspondem aos conectores, papéis de componentes representam os pontos de interface destes componentes (portas de componentes) e instâncias das subclasses de Participant correspondem aos papéis de conectores definidos pelo estilo arquitetural colaboração entre papéis (Seção 4.2).

Instâncias das subclasses de Collaboration tem referências para: (i) os participantes da colaboração, (ii) exceções externas e internas, (iii) colaborações aninhadas, (iv) a colaboração mãe (caso exista), (v) objetos (locais) usados na comunicação entre os participantes e (vi) objetos externos . Exceções internas são aquelas exceções que deveriam ser tratadas por todos os participantes da colaboração enquanto exceções externas são aquelas que deveriam ser propagadas para o nível superior do sistema. Subclasses de Collaboration devem redefinir o método ConfigSharedObjects que é responsável pela criação dos objetos compartilhados usados na comunicação inter-participantes. Para acessar estes objetos,

cada participante deve solicitar à correspondente colaboração referências a estes objetos através da invocação do método getSharedObject. Se uma colaboração possui colaborações aninhadas, o projetista desta aplicação deve redefinir o método ConfigNestedColl que é responsável pela criação dos objetos que representam as colaborações aninhadas. Para acessar os objetos que representam as colaborações aninhadas, cada participante deve solicitar à correspondente colaboração referências a estes objetos através da invocação do método getNestedColl.

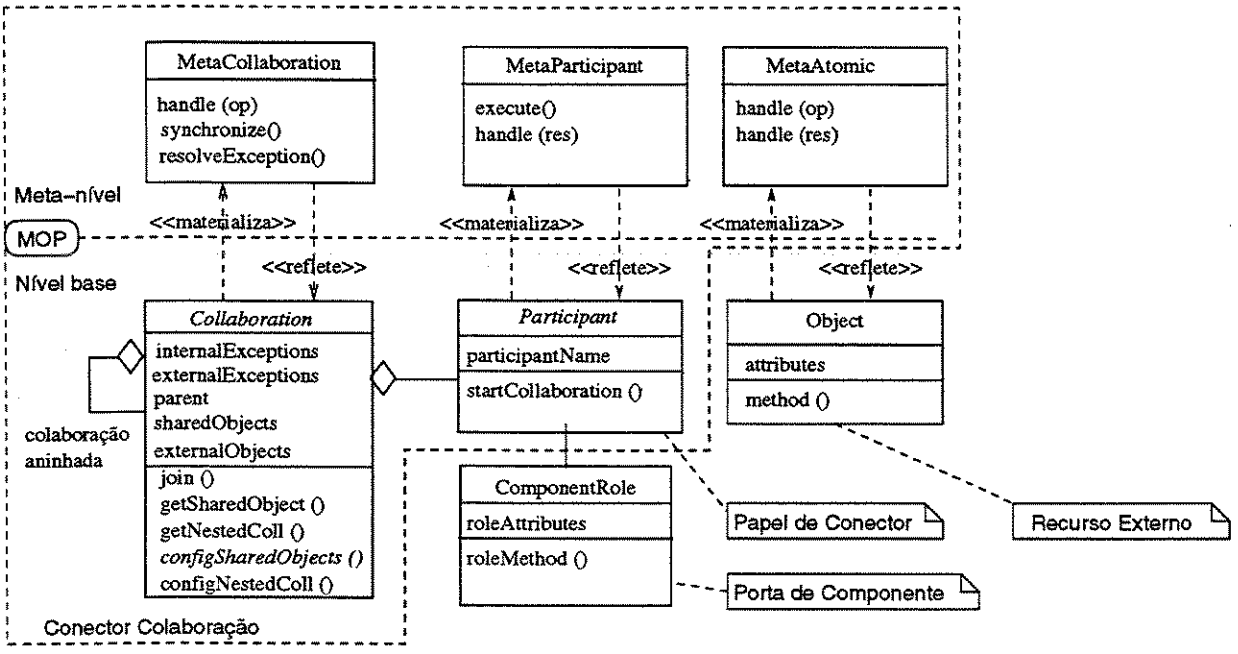


Figura 5.16: Padrão colaboração confiável

Instâncias de subclasses de Participant representam os participantes da colaboração e possuem referências para: (i) a colaboração que está participando; (ii) para o papel de componente (Seção 5.3.1); e (iii) para o método do papel de componentes que será executado durante a colaboração. Instâncias de MetaParticipant são associadas a instâncias das subclasses de Participant e são responsáveis por: (i) invocação do método do papel de componente; (ii) informar ao correspondente MetaCollaboration o término da execução do método; (iii) interceptar as exceções levantadas e informar ao correspondente MetaCollaboration; e (iv) após o processo de resolução ter terminado e a exceção resolvida ter sido retornada, invocar o correspondente tratador para esta exceção. Instâncias de MetaCollaboration estão associadas a instâncias de subclasses de Collaboration e são responsáveis por: (i) realizar a recuperação de erros. Tenta-se primeiro a recuperação por avanço (resolução de exceções + tratamento de exceções). Caso, não seja possível a recuperação por avanço, invoca-se o processo de restauração de estado (recuperação por retrocesso); e (ii)

sincronizar os participantes da colaboração.

Este padrão assegura que não existe nenhuma interação entre os participantes (papéis de componentes) e outros componentes que não fazem parte desta colaboração durante o período que a atividade cooperativa está sendo executada. Este padrão utiliza a solução de projeto definida pelo padrão **servidor de trancas**[2] [60].

O padrão **servidor de trancas** provê um controlado acesso concorrente a recursos compartilhados. O **servidor de trancas** é um componente independente que é separado do recurso compartilhado. Todos os clientes do recurso compartilhado sabem que antes de acessar o recurso, eles devem solicitar ao **servidor de trancas** uma tranca neste recurso. Após o cliente ter completado o seu acesso ao recurso ele deve liberar a tranca, de tal forma que todos os clientes possam acessar os recursos de forma justa. O padrão **colaboração confiável** solicita trancas em todos os papéis de componentes antes de iniciar a colaboração e libera estas trancas após a atividade cooperativa ter terminado. Desta forma, este padrão assegura que não existe nenhuma interação entre os participantes (papéis de componentes) e outros componentes que não fazem parte desta colaboração durante o período que a atividade cooperativa está sendo executada.

Como mencionado, este padrão emprega recuperação de erros por retrocesso, caso a recuperação de erros por avanço não tenha sucesso. Isto envolve o estabelecimento de pontos de recuperação, que são pontos durante a execução dos componentes para as quais os estados destes componentes podem ser posteriormente restaurados. Este padrão utiliza a solução de projeto definida pelo padrão de projeto **memento** [49].

O padrão de projeto **memento** captura e externaliza estados internos de objetos de tal forma que o objeto pode ser restaurado para aquele estado posteriormente. Nós utilizamos uma variante reflexiva deste, objetivando facilitar e tornar transparente a tarefa de salvar e restaurar os estados dos objetos durante o estabelecimento dos pontos de recuperação. Projetistas de aplicações não necessitam se preocupar em definir métodos que implementam tais tarefas. O padrão **colaboração confiável** estabelece pontos de recuperação (salva o estado dos papéis de componentes) antes de iniciar a atividade cooperativa. Caso algum erro seja detectado e a recuperação de erros por avanço não tenha sucesso, os estados dos papéis de componentes são restaurados para os estados salvos durante o estabelecimento dos pontos de recuperação.

A combinação dos padrões de projeto **memento** e **servidor de trancas** provê uma solução de projeto para implementar as semânticas transacionais nos recursos (compartilhados) externos. Recursos externos são objetos simples que estão associados a instâncias da classe MetaAtomic que garante as semânticas transacionais nestes objetos.

[2]Em inglês: *lock server pattern.*

Dinâmica

Os seguintes diagramas ilustram dois cenários típicos deste padrão:

Cenário I. O primeiro diagrama (Figura 5.17) ilustra um cenário onde nenhuma exceção é levantada e a colaboração é terminada com sucesso. Nesta figura, por motivos de simplificação, apresentamos apenas um dos participantes (Participant) da colaboração (Collaboration). Este participante está associado a um papel de componente (Role) que executa algumas operações em um objeto externo (Object). Collaboration e Participant estão associados respectivamente a MetaCollaboration e MetaParticipant que implementam os aspectos gerenciais da colaboração. Role está associado a um meta-objeto (Delegador) que compõe um conjunto de meta-objetos. Novamente, por motivos de simplificação apenas ilustramos um destes meta-objetos (MetaAtomic1). E por fim, Object está associado a MetaAtomic2.

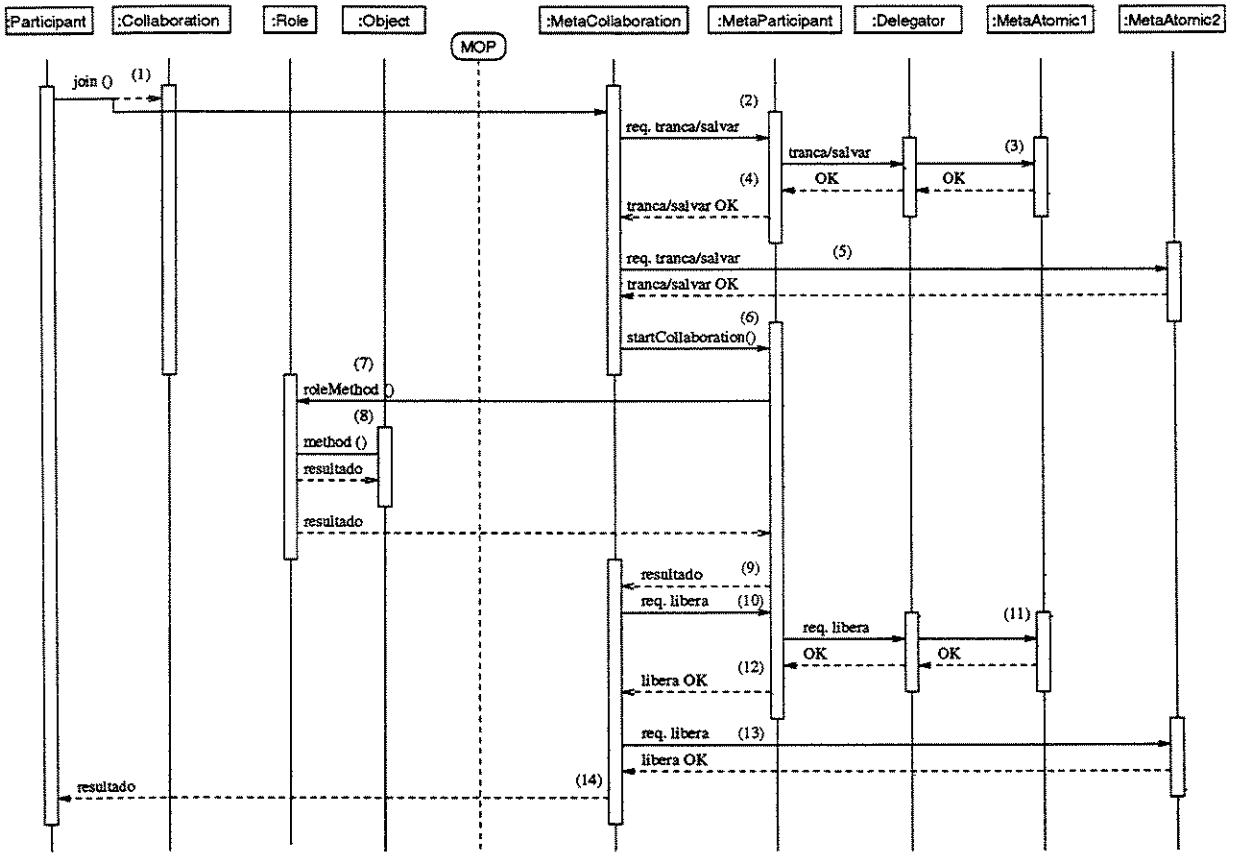


Figura 5.17: Dinâmica: padrão colaboração confiável (cenário I)

1. Participant informa que deseja iniciar a colaboração por invocar o método join de

Collaboration.

2. MetaCollaboration intercepta esta requisição e solicita ao MetaParticipant que este bloqueie e salve o estado do papel de componente a ele associado.

3. MetaParticipant então solicita a Delegador que este bloqueie e salve o estado do papel de componente a ele associado.

4. Delegador informa MetaParticipant (MetaCollaboration) que o papel de componente foi bloqueado e seu estado salvo.

5. MetaCollaboration então solicita a MetaAtomic2 que este bloqueie e salve o estado do objeto externo a ele associado.

6. Após receber a confirmação que o objeto externo foi bloqueado, MetaCollaboration solicita a MetaParticipant que inicie a sua atividade cooperativa.

7. Ao receber esta solicitação, MetaParticipant invoca roleMethod no papel de componente a ele associado.

8. Durante a execução de roleMethod, o papel de componente invoca method no objeto externo Object.

9. MetaParticipant intercepta o resultado da operação no papel de componente e retorna este resultado para MetaCollaboration.

10. MetaCollaboration então solicita a MetaParticipant que este desbloqueie o papel de componente a ele associado.

11. MetaParticipant então solicita a Delegador que este desbloqueie o papel de componente a ele associado.

12. Delegador informa o MetaParticipant (MetaCollaboration) que o papel de componente foi desbloqueado.

13. MetaCollaboration então solicita a MetaAtomic2 que este desbloqueie o objeto externo a ele associado.

14. Após receber a confirmação que o objeto externo foi desbloqueado, a atividade cooperativa termina.

Cenário II. O segundo diagrama (Figura 5.18) ilustra um cenário onde uma exceção é levantada, a recuperação de erros por avanço é realizada sem sucesso e a recuperação de

erros por retrocesso é realizada com sucesso. Novamente, por motivos de simplificação, apresentamos apenas um dos participantes (Participant) da colaboração (Collaboration). Este participante está associado a um papel de componente (Role) executa algumas operações em um objeto externo. Collaboration e Participant estão associados respectivamente a MetaCollaboration e MetaParticipant. Role está associado a um meta-objeto (Delegador) que compõe um conjunto de meta-objetos.

1. Participant informa que deseja iniciar a colaboração por invocar o método join de Collaboration.

2. MetaCollaboration intercepta esta requisição e solicita a MetaParticipant e MetaAtomic2 que estes bloqueiem e salvem os estados dos objetos a eles associados (Figura 5.17, passos 2-5).

3. Após receber a confirmação de que os objetos foram bloqueados, MetaCollaboration solicita a MetaParticipant que inicie sua atividade cooperativa.

4. Ao receber esta solicitação, MetaParticipant invoca roleMethod no papel de componente a ele associado.

5. Delegador intercepta esta operação e delega esta operação para MetaContract que o chama o teste de precondições e retorna noResult que informa o Delegador que ele pode delegar esta operação para o próximo meta-objeto na cadeia de delegação.

6. Delegador então delega a operação para MetaRole que invoca a operação no papel de componente presente no nível base. Durante a execução de roleMethod, o papel de componente invoca method no objeto externo Object. O resultado da execução de roleMethod é devolvido a Delegador.

7. Delegador recebe o resultado e delega para MetaContract que chama o teste de pós-condições devolvendo uma exceção desde que os testes de pós-condições não foram satisfeitos.

8. Delegador recebe a exceção e devolve esta como resultado da operação roleMethod. MetaParticipant intercepta os resultados das operações nos papéis de componentes e retorna o resultado para MetaCollaboration.

9. MetaCollaboration realiza a resolução de exceções e requisita MetaParticipant (Delegador) que este realize a recuperação de erros por avanço nos papéis de componentes.

10. Delegador delega esta requisição para MetaSearcher que invoca o tratador associado a exceção e devolve o resultado da execução do tratador. Neste exemplo, a execução do tratador é executada sem sucesso.

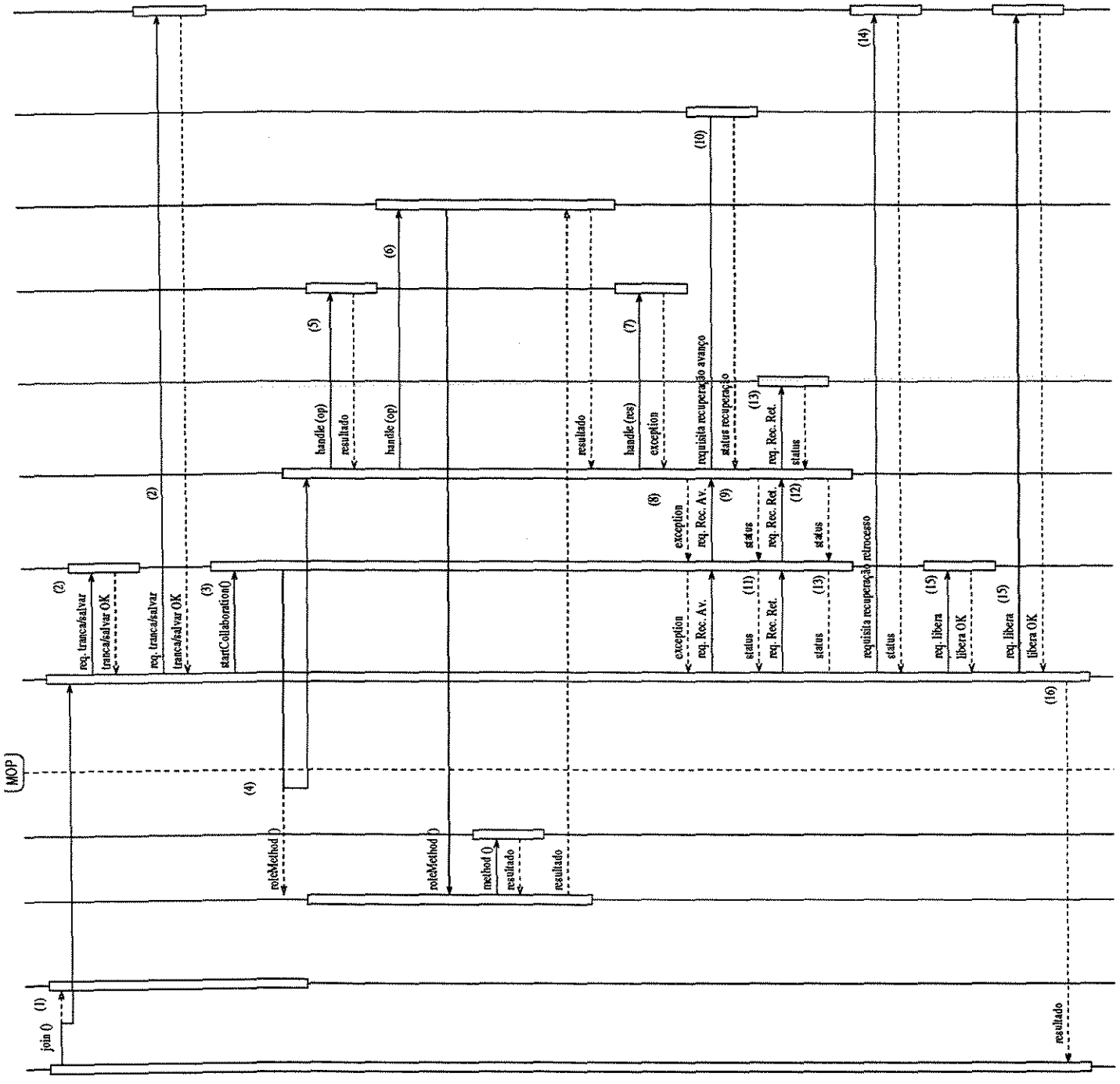


Figura 5.18: Dinâmica: padrão colaboração confiável (cenário II)

11. Delegador informa ao MetaParticipant (MetaCollaboration) que a recuperação de erros por avanço foi realizada sem sucesso.

12. MetaCollaboration então requisita MetaParticipant (Delegador) que este execute a recuperação de erros por retrocesso.

13. Delegador solicita a MetaAtomic1 que restaure o estado do papel de componente. Delegador informa MetaParticipant (MetaCollaboration) o status da recuperação.

14. MetaCollaboration requisita a MetaAtomic2 que este execute a recuperação de erros por retrocesso no objeto externo. MetaAtomic2 informa MetaCollaboration o status da recuperação.

15. MetaCollaboration solicita a MetaParticipant e MetaAtomic2 que estes desbloqueiem os objetos a eles associados (Figura 5.17, passos 10-13).

16. Após receber a confirmação que os objeto foram desbloqueados, a atividade cooperativa termina.

Variantes

- **Gerenciamento distribuído.** Nesta variante, o gerenciamento do grupo de participantes é distribuído. Desta forma, toda informação sobre o grupo deve ser mantida por cada participante. Conseqüentemente, as classes MetaCollaboration e Collaboration não são necessárias. O processo de resolução de exceções e sincronização é realizado de forma distribuída.

- **Gerenciamento não reflexivo.** Esta variante transfere todas as atividades gerenciais (resolução de exceções, sincronização, procura e invocação dos tratadores) para as classes do nível base Collaboration e Participant. Apesar de que tal variante não necessita utilizar um protocolo de meta-objetos para ser implementado, o código da aplicação possui invocações explícitas de tratadores, e procedimentos para resolução de exceções e sincronização. Conseqüentemente, o uso desta variante reduz a legibilidade da aplicação.

- **Nenhum acesso a recursos externos.** Nesta variante não existe acesso a recursos (compartilhados) externos. Conseqüentemente, as semânticas transacionais nestes recursos não necessita ser assegurada e portanto, a classe MetaAtomic é desnecessária.

Usos conhecidos

- Romanovsky *et al.* [103] utiliza uma variante não reflexiva e distribuída. Este trabalho propõe um algoritmo para a resolução de exceções concorrentes em sistemas orientados a objetos distribuídos. Resolução de exceções e sincronização é realizada de forma distribuída e a informação relacionada à atividade cooperativa deve ser mantida por cada participante. Cada participante mantém uma cópia do algoritmo e o gerenciamento é realizado através de troca de mensagens.

- Zorzo *et al.* [136] propôs um *framework* para a implementação de ações atômicas coordenadas (Seção 2.5.3) que provê projetistas um conjunto de classes que auxiliam a estruturação de suas aplicações. Este trabalho utiliza a variante não reflexiva deste padrão. Projetistas estendem duas classes deste *framework* (adicionando características específicas de suas aplicações) na definição das atividades cooperativas concorrentes. Ambas classes são similares às classes Collaboration e Participant em relação às suas responsabilidades.

- Conversações [67, 68, 100, 109] utilizam a variante sem acesso a recursos externos.

Conseqüências

- **Uniformidade.** A estratégia para tratamento de exceções concorrentes é uma extensão consistente da estratégia genérica para tratamento de exceções (Seção 5.1.2).

- **Transparência e simplicidade.** Mecanismos gerenciais para a recuperação de erros são realizadas de forma transparente para a aplicação. Programadores apenas precisam se preocupar na identificação das atividades cooperativas da aplicação.

- **Legibilidade e reutilização.** O código da aplicação não é entrelaçado com invocação de métodos responsáveis pela sincronização e resolução de exceções. Como conseqüência, ele aumenta a legibilidade da aplicação, que por sua vez aumenta a sua reutilização.

5.5 Um *framework* para o desenvolvimento de aplicações confiáveis

Os padrões de projeto discutidos neste capítulo foram utilizados na implementação de um *framework* orientado a objetos que provê uma infra-estrutura genérica para o desenvolvimento de sistemas concorrentes confiáveis. O *framework* é reflexivo, portanto, ele possui componentes presentes no nível base e no meta-nível. Projetistas utilizam este *framework* através da: (i) definição de novas classes (subclasses) necessárias para uma aplicação específica (Figura 5.19(a)) ou (ii) associação de objetos da aplicação a instâncias das meta-classes definidos pelo *framework* (Figura 5.19(b)).

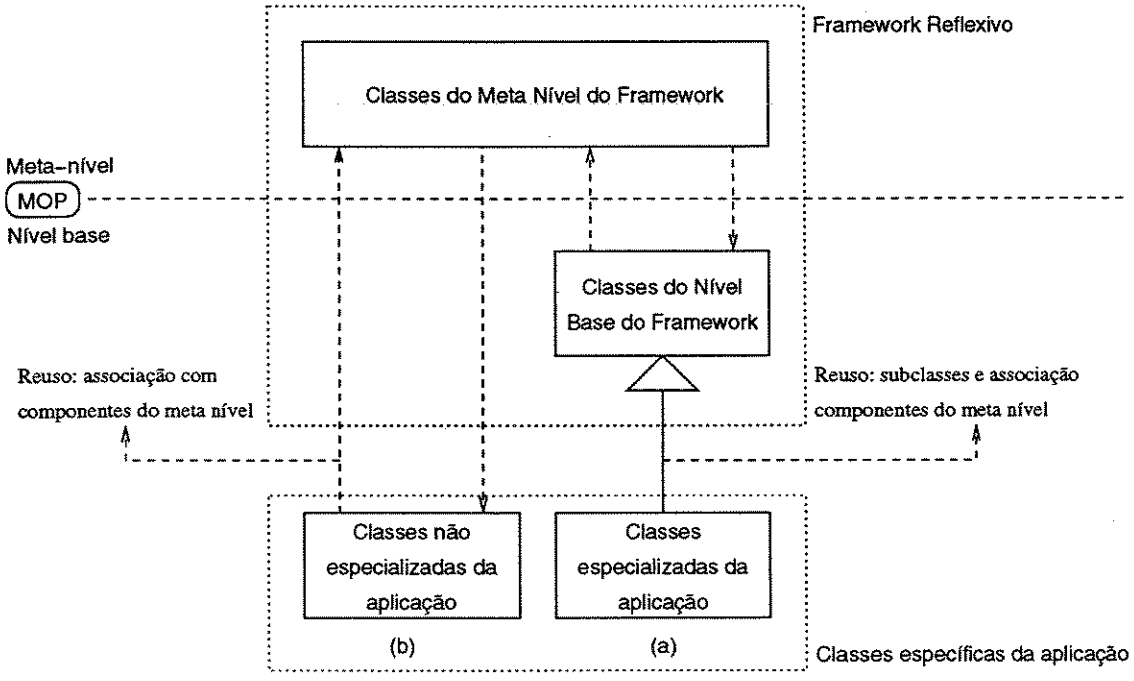


Figura 5.19: *Framework* reflexivo

A associação entre objetos do nível base e meta-objetos é feita de forma automática desde que os projetistas forneçam alguns arquivos de configuração que são lidos pelo *framework* e que inicializam a meta-configuração associada com uma aplicação específica. Aplicações são executadas através da invocação do método Main de uma classe especial do *framework* passando o nome da aplicação como parâmetro. Este método lê os arquivos de configuração, inicializa a meta-configuração associada e invoca a aplicação. O formato destes arquivos de configuração estão descritos no Apêndice A.

Nós utilizamos a linguagem de programação Java e um protocolo de meta-objetos chamado Guaraná [92, 93] na implementação deste *framework*. Em Guaraná, cada objeto

pode está diretamente associado com zero ou um meta-objeto. Ele permite diversos meta-objetos serem combinados em uma meta-configuração dinamicamente modificável. Deste modo, a composição de serviços não-funcionais é facilitada. Composers [93] são meta-objetos que combinam os comportamentos de outros meta-objetos. Então, um objeto pode ser diretamente associado a um meta-objeto (composer) que encapsula diversas propriedades não-funcionais.

Capítulo 6

Um Método para o Desenvolvimento de Sistemas Confiáveis

Nos capítulos anteriores, apresentamos uma arquitetura de software para o desenvolvimento de aplicações confiáveis e um conjunto coeso de padrões que refinam os componentes lógicos e conectores da arquitetura de software proposta.

No entanto, não basta apenas definir a arquitetura. É necessário definir um conjunto de diretrizes, com razoável nível de detalhe, guiando projetistas na incorporação de características de tolerância a falhas durante o ciclo de desenvolvimento do software, isto é, iniciando-se na fase de projeto e terminando na fase de codificação/implementação do software. Neste capítulo, apresentamos um método para o desenvolvimento de sistemas confiáveis que contém este conjunto de diretrizes.

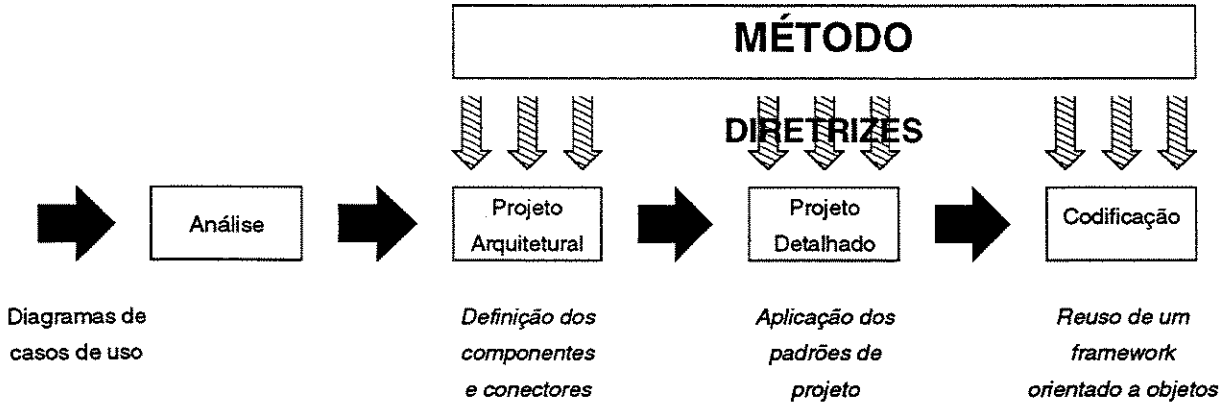


Figura 6.1: Método para o desenvolvimento de sistemas confiáveis

Vale a pena salientar que o método descrito neste capítulo apenas guia as fases de projeto e codificação do ciclo de desenvolvimento de sistemas confiáveis e é assumido que as

fases de levantamento de requisitos e de análise do sistema já foram realizadas. Como discutido, nós adotamos neste trabalho UML como a linguagem de modelagem de sistemas, que produz melhores resultados se é utilizado um processo de desenvolvimento orientado a casos de uso, centrado numa determinada arquitetura, interativa e incremental. Desta forma, nós assumimos que durante as fases de identificação de requisitos e de análise, a identificação das propriedades funcionais do sistema se dá a partir da criação de cenários de casos de uso que descrevem as atividades básicas do sistema. Além disso, associado a cada caso de uso, são identificados os cenários de falhas e as possíveis exceções levantadas pela manifestação de falhas.

6.1 Descrição do método

Um método é uma maneira de fazer algo de forma ordenada [56]. Consideramos que um método compõe-se de atividades que definem as diretrizes a serem seguidas para a obtenção do resultado, neste caso específico, um sistema orientado a objetos confiável. As atividades podem ser formadas por um conjunto de tarefas interconectadas, onde as saídas de uma tarefa funcionam como entradas de tarefas subseqüentes.

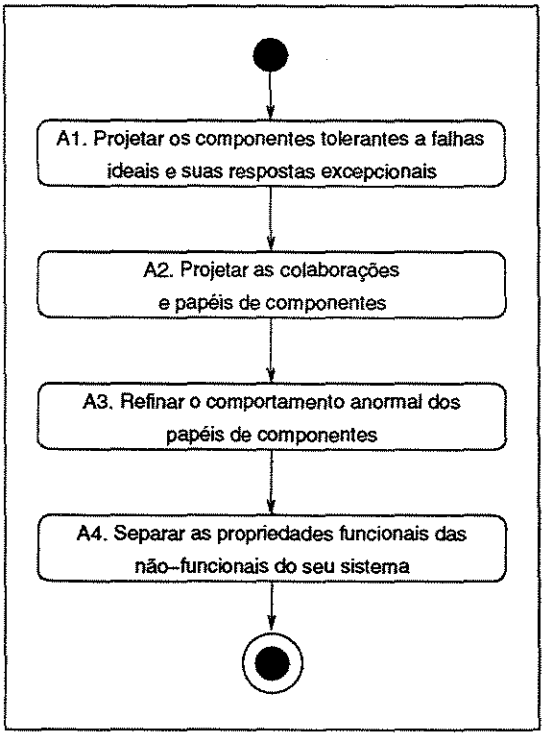


Figura 6.2: Atividades do Método

Assim, para fornecer uma visão geral do método, apresentamos a sua seqüência de atividades (Figura 6.2) representadas por um diagrama de atividades, expresso na notação UML. As tarefas associadas a cada atividade serão introduzidas nesta seção.

6.2 Atividade A1. Projetar os componentes e suas respostas excepcionais

O objetivo desta atividade é projetar os componentes tolerantes a falhas ideais do sistema e suas respostas excepcionais. Esta atividade é subdividida em 6 tarefas, T1 a T6, que são explicadas a seguir.

Tarefa T1. Utilizar contratos na especificação dos componentes

A especificação de um componente tolerante a falhas ideal não é uma tarefa trivial. Os seus serviços devem ser confiáveis, ou seja, eles devem ser executados corretamente e retornar um resultado esperado, ou devem retornar uma exceção sinalizando que o serviço não pode ser executado normalmente. Desta forma, a especificação deve incluir não apenas os requisitos funcionais dos componentes, que correspondem ao seu comportamento normal, mas também todos os aspectos excepcionais que podem ocorrer. Deve-se, portanto, especificar detalhadamente todas as precondições que devem ser atendidas para que um serviço seja executado normalmente, as pós-condições que são esperadas pelos clientes que utilizam o serviço, e a invariantes do componente, que são as restrições sobre o estado do componente que devem ser respeitadas na execução de qualquer serviço. Qualquer violação destas condições deve gerar uma resposta excepcional na execução de um serviço.

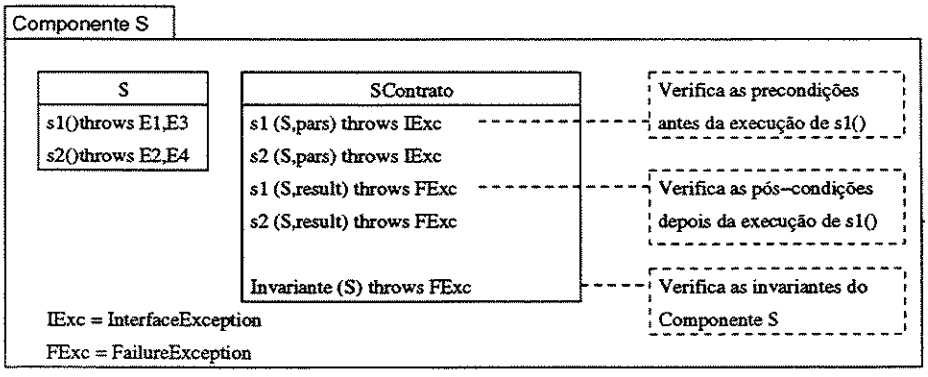


Figura 6.3: Contratos na especificação de componentes

Neste contexto, projetistas de aplicações deveriam empregar o padrão de projeto **contrato reflexivo** (Seção 5.1.1) que proporciona suporte à utilização da metodologia projeto por contratos [88] na estruturação das atividades de detecção de erros dos componentes de software do sistema. Figura 6.3 ilustra o projeto do componente S que recebe requisições de serviços e produz respostas. O projeto deste componente é realizada através da definição de duas classes: a classe S que implementa os requisitos funcionais deste componente e a classe SContrato que implementa os testes de precondições, pós-condições e invariantes deste componente.

Tarefa T2. Separar as atividades normais e excepcionais dos componentes

A separação explícita entre os comportamentos normais e excepcionais melhora diversos aspectos de qualidade do sistema, tais como legibilidade, facilidade de manutenção e reutilização.

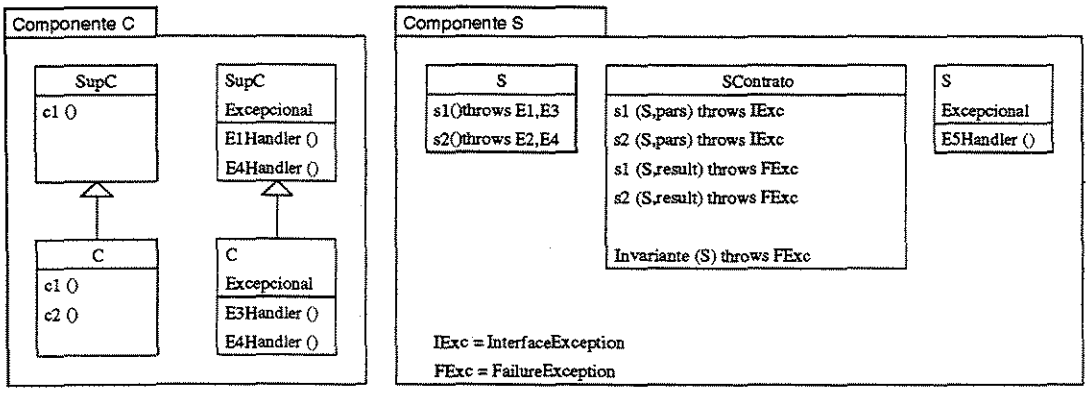


Figura 6.4: Hierarquia de classes normais e excepcionais

Neste contexto, projetistas de aplicações orientadas a objetos deveriam empregar os padrões de projeto **estratégia de tratamento de exceções** (Seção 5.1.2) e **tratador** (Seção 5.1.3) e criar um conjunto de classes normais que implementam as atividades normais dos componentes de software, e classes excepcionais que implementam as atividades anormais (Figura 6.4). Conseqüentemente, existe uma separação clara entre as atividades normais e excepcionais de uma aplicação. Figura 6.4 ilustra o projeto do componente C que requisita serviços para o componente S. O componente C deveria ser apto a lidar com as respostas normais e principalmente as excepcionais do componente S. Na Figura 6.4, os métodos da classe excepcional ExceptionalSupC são os tratadores das exceções que deveriam ser tratadas pelos métodos da classe SupC. Projetistas podem compor uma hierarquia de classes excepcionais que é ortogonal à hierarquia de classes normais da aplicação. As classes excepcionais ExceptionalSupC e ExceptionalC são organizadas hierarquicamente de

tal forma que a hierarquia resultante é ortogonal à hierarquia de classes normais (SupC e C). Hierarquias de classes excepcionais permitem subclasses excepcionais herdarem tratadores de suas superclasses e, conseqüentemente, eles permitem reutilização de código excepcional.

Tarefa T3. Representar exceções como classes

Exceções deveriam ser representadas como classes. Diferentes tipos de exceções são organizadas hierarquicamente como classes. A classe Exception é a raiz desta hierarquia. Alguns mecanismos de tratamento de exceções existentes adotam esta abordagem, tais como [21, 22, 32, 39]. O padrão de projeto **exceção** (Seção 5.2.1) provê suporte para a especificação das exceções que podem ser levantadas durante a execução do sistema. Diferentes tipos de exceções são organizadas hierarquicamente como classes que derivam da classe raiz Exception (Figura 6.5).

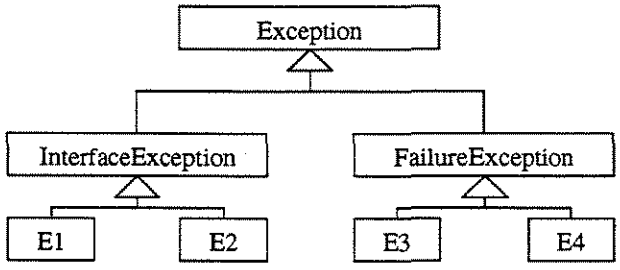


Figura 6.5: Exceções de interface e de defeito

De acordo com o modelo de componente tolerante a falhas ideal (Seção 2.1.3), existem dois tipos de respostas excepcionais que podem ser retornadas por um serviço: exceção de interface, que indica que as precondições deste serviço foram violadas, e exceções de defeito, que indicam que o componente não pode fornecer os seus serviços normalmente (as pós-condições foram violadas). Todos os serviços devem prever a sinalização de, no mínimo, estes tipos de exceções mais básicos, assim os componentes clientes devem, no mínimo, prever o tratamento para estas exceções. Respostas excepcionais que podem ser sinalizadas por um método deveriam ser descritas na assinatura do método. Figura 6.4 ilustra que o método s2 pode sinalizar as exceções E2 e E4.

Tarefa T4. Associar tratadores em multi-níveis

Com objetivo de melhorar a estruturação de sistemas de software, é desejável permitir alguma flexibilidade em relação à associação de tratadores de exceções. Projetistas de aplicações orientadas a objetos deveriam organizar seus sistemas por aplicar a associação

de tratadores em multi-níveis, isto é, a associação de tratadores a regiões protegidas de diversos níveis tais como classes, objetos, métodos etc. Primeiramente, tratadores deveriam ser associados a exceções. Tratadores também podem ser associados a uma classe. Na Figura 6.4, os métodos da classe SupCExcepcional são tratadores de classes para as exceções que deveriam ser tratados no contexto dos métodos da classe SupC. Analogamente, métodos da classe ExcepcionalS são tratadores de classes para as exceções que deveriam ser tratadas no contexto da execução dos métodos da classe S. Por outro lado, os tratadores de classes para as exceções que deveriam ser tratadas no contexto da execução dos métodos da classe C podem ser métodos da classe ExcepcionalC ou métodos que são herdados das superclasses de ExcepcionalC. Conseqüentemente, o tratador para a exceção E1 (E1Handler) é herdado da classe ExcepcionalSupC.

Em adição, tratadores associados a objetos podem também ser definidos. Para implementar tratadores associados a objetos individuais, uma nova classe excepcional deveria ser criada. Esta nova classe implementa os tratadores de objetos para as exceções que deveriam ser tratadas no contexto de qualquer método do objeto.

A procura para os tratadores para as exceções levantadas deveria ser a seguinte: (i) se existe uma classe excepcional associada ao objeto, o mecanismo tenta encontrar o tratador associado ao método que levantou a exceção; (ii) se nenhum é encontrado, o sistema tenta encontrar tratadores em classes excepcionais ou superclasses destas associadas à classe normal cujo objeto é uma instância; (iii) se nenhum destes é encontrado, a exceção é propagada para o objeto que invocou o método e os passos (i) e (ii) são repetidos; (iv) se ainda nenhum tratador foi encontrado, o sistema procura por tratadores associados à própria exceção levantada.

Tarefa T5. Utilizar propagação explícita de exceções e o modelo de terminação

A adoção de explícita propagação de exceções (Seção 2.3.2) tem um número de benefícios [31, 133]. Nesta abordagem, o tratamento da exceção é limitado para o chamador imediato. Entretanto, a exceção pode ser resinalizada explicitamente no contexto de um tratador para um componente de mais alto nível. Apesar dos ganhos na simplicidade de programação, o uso de propagação automática de exceções permanece sujeito a falhas dado que eles são as partes menos documentadas e testadas de uma interface [32].

Desenvolvedores de sistemas orientados a objetos confiáveis deveriam escolher apenas o modelo de terminação (Seção 2.3.4) que consiste em terminar a execução da unidade que levantou a exceção e então transferir o fluxo de controle para o tratador da exceção. A semântica do modelo de terminação é mais simples e mais adequado para a construção de sistemas confiáveis. O modelo de continuação é muito poderoso e flexível, mas por sua vez é difícil de ser utilizado por programadores de aplicações. Em fato, eles podem

promover a prática de programação não segura de remover o sintoma de um erro sem remover a sua causa.

Tarefa T6. Utilizar diversidade no projeto de seus componentes

Para construir software que trata falhas de seus componentes, nós necessitamos de técnicas que lidam com falhas de projeto. As técnicas de tolerância a falhas de software são baseadas no uso de redundância. Entretanto, a simples replicação de componentes não é suficiente para lidar com falhas de projeto. Por essa razão, a diversidade de projeto é de fundamental importância e significa que os componentes são baseados em diferentes projetos. Neste contexto, projetistas de aplicações deveriam utilizar o padrão de projeto **redundância de software** (Seção 5.1.4) que apresenta uma solução uniforme para a incorporação de diversidade de projeto em um sistema orientado a objetos. Figura 6.6 ilustra o projeto do componente S que contém duas versões redundantes para os serviços fornecidos por este componente.

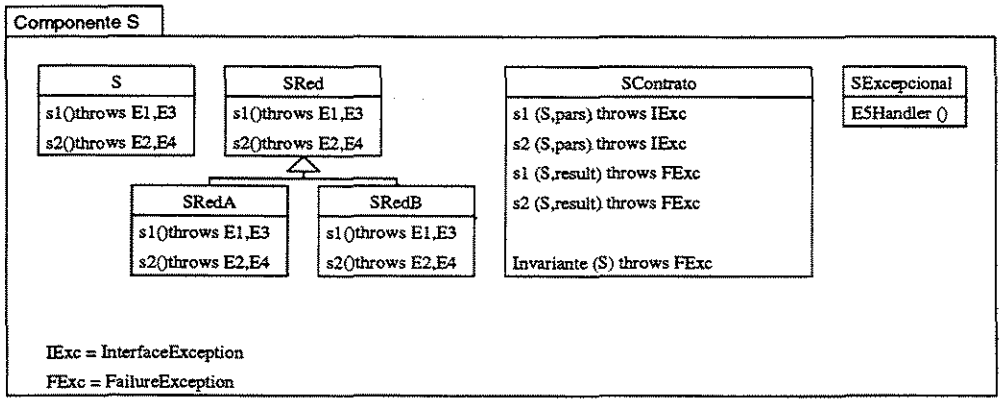


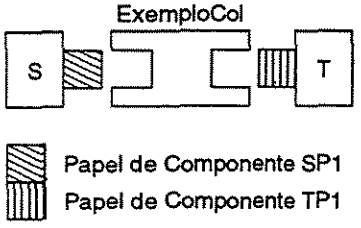
Figura 6.6: Diversidade de projeto

6.3 Atividade A2. Projetar as colaborações e os papéis de componentes

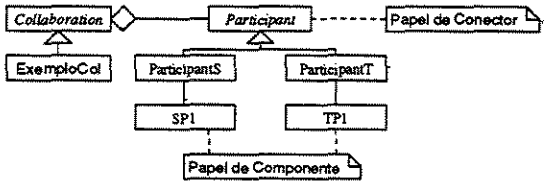
O objetivo desta atividade é projetar as colaborações confiáveis e os papéis de componentes.

Como discutido na Seção 4.4, nós assumimos uma estrutura em camadas e que componentes tolerantes a falhas ideais presentes na mesma camada podem realizar atividades

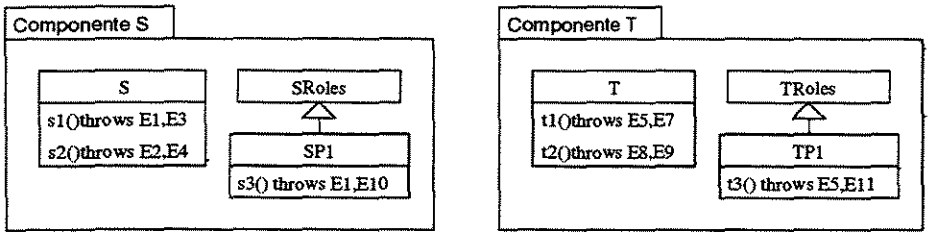
cooperativas, cada uma desempenhando um específico papel de componente em uma colaboração. Projetistas deveriam utilizar os padrões de projeto **papel reflexivo** (Seção 5.3.1) que apresenta soluções de projeto para a implementação de papéis de componentes e **colaboração confiável** (Seção 5.4.1) que provê soluções de projeto para a implementação do protocolo de interação que deve ser seguido pelos componentes envolvidos em uma colaboração.



(a) Colaboração entre papéis



(b) Projeto detalhado da colaboração



(c) Projeto detalhado dos papéis de componentes

Figura 6.7: Colaborações e papéis de componentes

Figura 6.7(a) ilustra uma colaboração entre dois componentes tolerantes a falhas ideais. Nesta figura, o componente S está desempenhando o papel de componente SP1 enquanto que o componente T está desempenhando o papel de componente TP1. Figura 6.7(b) ilustra a utilização do padrão de projeto **colaboração confiável** (Seção 5.4.1) no projeto detalhado desta colaboração enquanto que a Figura 6.7(c) ilustra a utilização do padrão de projeto **papel reflexivo** (Seção 5.3.1) no projeto detalhado dos papéis de componentes.

6.4 Atividade A3. Refinar os papéis de componentes

Após os projetistas terem identificado as colaborações e os respectivos papéis de componentes, eles necessitam refinar os papéis de componentes objetivando incluir os mecanismos de detecção de erros e tratadores para as exceções que deveriam ser tratadas

cooperativamente no contexto de uma colaboração. Vale a pena salientar que as tarefas T7-T10 que compõem esta atividade são refinamentos das tarefas T1, T2, T3 e T6 descritas anteriormente.

Tarefa T7. Utilizar contratos na especificação dos papéis de componentes

Projetistas deveriam empregar o padrão de projeto **contrato reflexivo** (Seção 5.1.1) na estruturação de detecção de erros que devem ser tratadas cooperativamente no contexto da colaboração.

Na Figura 6.8, SP1Contrato estende SContrato e implementa as atividades de detecção de erros no contexto da colaboração ExemploCol mostrado na Figura 6.7(a). As classes contrato SContrato e SP1Contrato são organizados hierarquicamente de tal forma que a hierarquia resultante é ortogonal à hierarquia de classes normais e de papéis (S e SP1). Hierarquias de classes contrato permitem às subclasses herdarem detectores de erros de suas superclasses e, conseqüentemente, eles permitem a reutilização de detectores de erros.

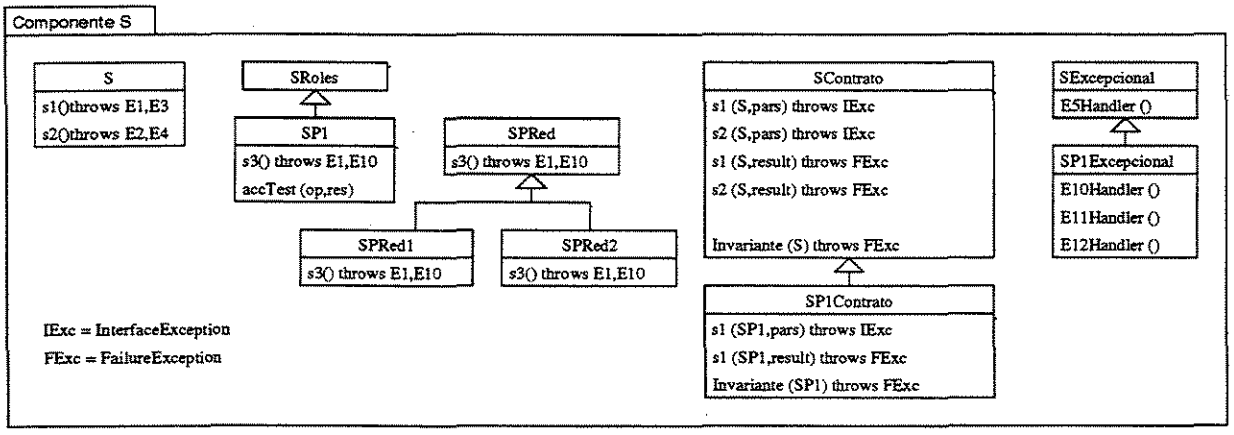


Figura 6.8: Refinamento do componente S

Tarefa T8. Separar as atividades normais e excepcionais dos papéis de componentes

Projetistas deveriam empregar o padrões de projeto **estratégia de tratamento de exceções** (Seção 5.1.2) e **tratador** (Seção 5.1.3) na estruturação dos tratadores de exceções que devem ser tratadas cooperativamente no contexto da colaboração.

Na Figura 6.8, SP1Excepcional estende SExcepcional e implementa as atividades de tratamento de exceções no contexto da colaboração ExemploCol. As classes excepcionais SExcepcional e SP1Excepcional são organizados hierarquicamente de tal forma que a hierarquia resultante é ortogonal à hierarquia de classes normais e de papéis (S e SP1).

Hierarquias de classes excepcionais permitem subclasses herdarem tratadores de exceções de suas superclasses e, conseqüentemente, eles permitem a reutilização de código excepcional.

Tarefa 9. Representar as exceções concorrentes como classes

Projetistas deveriam empregar o padrão de projeto **exceção** (Seção 5.2.1) que provê suporte à definição das exceções simples e estruturadas que podem ser levantadas pelos participantes da colaboração. Vale a pena relembrar que as exceções concorrentes são exceções de defeito levantadas pelos componentes tolerante a falhas ideais. Figura 6.9 ilustra o refinamento das respostas excepcionais que incluem as exceções concorrentes levantadas pelos papéis de componentes durante o contexto de uma colaboração.

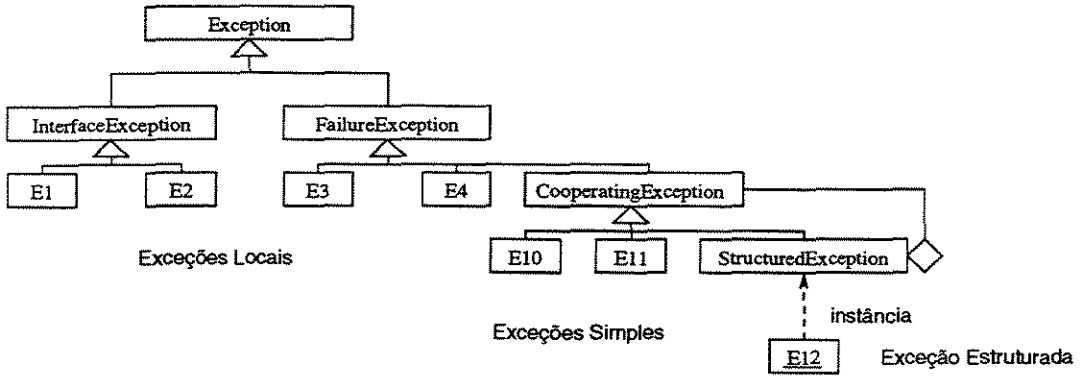


Figura 6.9: Refinamento das respostas excepcionais

O padrão de projeto **exceção** (Figura 6.9) provê suporte à definição das exceções simples e estruturadas que podem ser levantadas pelos participantes da colaboração.

Tarefa 10. Utilizar diversidade no projeto dos papéis de componentes

Projetistas deveriam empregar o padrão de projeto **redundância de software** (Seção 5.1.4) no projeto redundante dos papéis de componentes. Na Figura 6.8, SP1Red (e suas subclasses) ilustra o projeto do papel de componente SP1 que contém duas versões redundantes para os serviços providos por este papel de componente.

6.5 Atividade A4. Separar as propriedades funcionais das não-funcionais de seu sistema

Projetistas deveriam utilizar o *framework* orientado a objetos (Seção 5.5) na codificação de suas aplicações. Este *framework* provê suporte a uma separação explícita das propriedades funcionais e não-funcionais do sistema. Projetistas podem empregar a noção de separação de interesses e se concentrar na definição da funcionalidade de suas aplicações, abstraindo da implementação das propriedades não-funcionais.

6.6 Considerações finais

Neste capítulo apresentamos um método para o desenvolvimento de sistemas confiáveis. Este método contém um conjunto de diretrizes que guia projetistas na incorporação de características de tolerância a falhas durante o ciclo de desenvolvimento do software, iniciando-se na fase de projeto e terminando na fase de codificação/implementação do software.

Nos próximos dois capítulos são apresentados três estudos de caso com o intuito de ilustrar a aplicabilidade da utilização do método descrito neste capítulo no projeto e implementação de sistemas orientados a objetos confiáveis.

Capítulo 7

Estudo de Caso: Controle de Estações Ferroviárias

Este capítulo descreve o estudo de caso **controle de estações ferroviárias** e mostra como os estilos arquiteturais e padrões de projeto discutidos nos capítulos 4 e 5 foram empregados no projeto deste estudo de caso [12, 11]. Este estudo de caso está focado no desenvolvimento de um sistema de controle ferroviário que lida com o controle e a coordenação de trens nas proximidades de estações ferroviárias. Estações possuem diversas plataformas onde trens podem parar (com uma restrição óbvia que no máximo um trem pode parar em uma estação por vez). Nós assumimos que trens transportam passageiros de uma estação origem para uma estação destino e que trens podem executar uma atividade cooperativa quando eles param juntos em uma mesma estação ferroviária, por exemplo, passageiros podem trocar de trem com o objetivo de tornar a sua viagem mais rápida.

7.1 Descrição

O programa de controle de estações ferroviárias deve satisfazer certas condições, tais como:

- Segurança contra acidentes[1]: (i) colisões entre trens devem ser prevenidas. Logo, uma distância segura entre trens deve ser mantida.

- Justiça[2]: trens param nas estações de forma justa. Isto é, existe justiça na alocação das plataformas onde os trens param.

[1] Em inglês: *safety*.
[2] Em inglês: *fairness*.

- Vivacidade[3]: a garantia de que todos os passageiros cheguem aos seus respectivos destinos não é responsabilidade das estações e sim de todo o sistema de controle ferroviário. Estações apenas provém informações para um componente que faz o escalonamento dos passageiros. Estações apenas obedecem tais escalonamentos.

7.1.1 Atuadores e sensores

- Atuadores influenciam o sistema através dos seguintes comandos: (i) mudar a direção dos conectores de trilho presentes na malha ferroviária; (ii) iniciar/parar trens; (iii) aumentar/diminuir a velocidade dos trens; e (iv) inverter a direção do movimento de trens.

- Nós assumimos que trilhos ferroviários possuem sensores que fornecem ao sistema de controle, informação suficiente sobre a posição dos trens e conseqüentemente sobre o estado do sistema. É de extrema importância que a integridade da informação fornecida pelos sensores seja checada e verificada, pois esta informação não é usada apenas para manter atualizada a posição dos trens mas principalmente para evitar colisões entre trens.

7.1.2 Modelo de falhas

Este estudo de caso emprega técnicas para detectar erros causados por falhas (por exemplo, um trem não consegue parar), para recuperar estes erros e restaurar a execução normal do sistema. Antes de definir e analisar as possíveis falhas, nós apresentamos suposições adicionais sobre o comportamento do sistema. As suposições apresentadas nesta seção baseiam-se no modelo de falhas descrito em Rubira [105]:

- O relógio do sistema nunca falha;

- Informações de sensores e do relógio do sistema são sempre transmitidas corretamente sem qualquer perda ou erro; e

- Apenas uma falha pode acontecer nos sensores e conectores presentes em um trilho ou no trem usando aquele trilho durante o intervalo de interesse.

[3]Em inglês: *liveness*

Falhas temporais

Este estudo de caso deve satisfazer a seguinte restrição temporal: Um trem não pode chegar em uma estação após t_{Entry} e deixar a estação antes de t_{Exit}.

Se um trem falha em chegar antes de t_{Entry}, então os outros trens continuam as suas atividades porém algumas ações corretivas devem ser tomadas, por exemplo, alguns passageiros podem ser deixados na estação esperando o trem que está atrasado. Quando o trem atrasado chega na estação, alguns de seus passageiros podem ter perdido as suas conexões; conseqüentemente um novo escalonamento destes passageiros deve ser feita.

Falhas de sensores

Sensores são dispositivos que detectam a presença de trens e representam a única fonte de informação sobre o estado da malha ferroviária. Porém, sensores não são dispositivos confiáveis, pois às vezes podem ser ativados erroneamente. Um sensor falha de dois modos: (i) quando ele detecta a presença de um trem quando ele não deveria. Isto é, o sensor falho não faz parte da rota de nenhum trem durante o intervalo de interesse; e (ii) quando ele não detecta a presença de trens quando ele deveria. Tal falha é identificada quando um diferente mas previsível sensor acusa a presença do trem.

Falhas de atuadores

Estas falhas estão relacionadas com situações onde um trem ou um conector de trilho falha.

- **Falhas nos conectores de trilho.** Se um conector de trilho está quebrado, é muito provável que ele seja posto em uma direção e se recuse a mudar desta direção. É também possível que o conector de trilho não mude sua direção em resposta a uma requisição, mas repetindo a requisição pode resolver o problema (falhas intermitentes).

- **Falhas nos trens.** Novamente a falha pode ser permanente ou intermitente, e repetir a requisição pode solucionar o problema.

  - Se um trem falha em se mover após ter parado em uma estação não acarreta nenhum prejuízo sério, exceto que algumas restrições temporais não serão cumpridas. Esta falha não afeta apenas o trem falho (e seus passageiros), mas todos os outros trens que necessitam parar na plataforma ocupada pelo trem falho.

– A situação mais desastrosa é um trem falhar em parar. Neste caso, obviamente, o trem falho não respeitará sua rota. A única recuperação para esta falha é direcionar o trem para uma região segura da malha ferroviária, enquanto assegurando que os outros trens são parados se existe possibilidade de colisão.

7.2 Projeto arquitetural

O objetivo desta seção é mostrar a praticabilidade de utilizar os estilos arquiteturais (Capítulo 4) e o método (Capítulo 6) discutidos no projeto arquitetural deste estudo de caso. Nós assumimos uma estrutura em camadas do sistema de controle e que trens executam uma atividade cooperativa quando eles param juntos em uma mesma estação ferroviária.

Nosso projeto separa os requisitos funcionais da aplicação entre um conjunto de colaborações que ocorrem durante a execução do sistema e um conjunto de controladores de trens que determinam as rotas dos trens e conseqüentemente a ordem em que as colaborações são executadas. Nós assumimos que o requisito de segurança contra acidentes é garantido pela camada subjacente (em nossos experimentos esta camada é representada pela **camada malha ferroviária** [95, 105]) em cima da qual nós construímos os controladores de trens e as atividades cooperativas entre trens (Figura 7.1). Ademais, meta-objetos são responsáveis pela implementação das propriedades não-funcionais dos componentes.

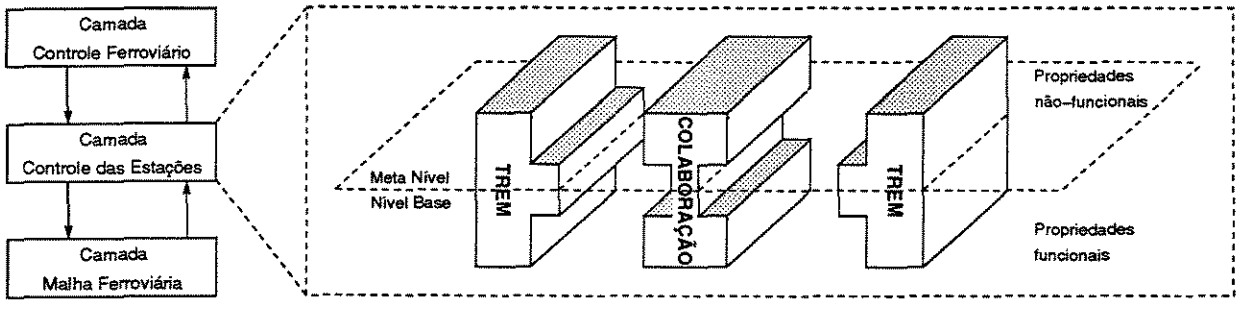


Figura 7.1: Projeto arquitetural do estudo de caso

Nós assumimos que cada passageiro tem um bilhete que descreve o trajeto de sua viagem. Se alguma falha afeta este trajeto, o componente escalonador pode ser usado para recalculá-lo. Este componente faz parte da **camada controle ferroviário**. Plataformas são modeladas como recursos externos e têm salas de espera onde passageiros esperam por seus trens. As colaborações entre os trens implementam recuperação de erros coordenada e mantêm a consistência destes recursos externos na presença de falhas e concorrência

entre diferentes atividades competindo por estes recursos.

7.3 Projeto detalhado

O objetivo desta seção é apresentar o projeto detalhado das três camadas ilustradas na Figura 7.1. Um desafio do projeto deste estudo de caso é ser tão realista quanto possível mas ao mesmo tempo ter condições de checar a validade de nosso projeto. Desta forma, nós utilizamos um controle de uma malha ferroviária disponível na universidade de Newcastle (Inglaterra) [105, 114] para implementar nossas idéias.

7.3.1 Camada: malha ferroviária

A malha ferroviária é montada em partes separadas, que são controladas por computadores independentes, ligados por uma rede de computadores. Cada instância da classe Controlador da Malha controla uma parte da malha ferroviária e corresponde, em tempo de execução, a um componente distribuído na rede de computadores. Cada instância da classe Controlador da Malha usa uma instância da classe Protocolo de Comunicação para se comunicar com outra instância de Controlador da Malha presente em uma máquina remota. A classe Protocolo de Comunicação implementa a funcionalidade básica para comunicação entre dois controladores distribuídos e, portanto, encapsula detalhes de comunicação, tais como, estabelecimento da comunicação, empacotamento de argumentos e desempacotamento de resultados. Cada parte da malha ferroviária é composta por conectores de trilho, sensores e trilhos (que ligam conectores e sensores). A classe Malha é composta por um conjunto de seções, as quais são compostas por regiões (isto é, sensores) e conectores de trilho (Figura 7.2).

Uma seção é uma ligação orientada entre duas regiões adjacentes. Cada seção, delimitada por duas regiões adjacentes, pode conter uma seqüência de zero ou mais conectores de trilho (Figura 7.3). Conectores de trilho e regiões são ligados entre si através de arestas, que são partes do trilho da malha. A direção de uma seção S é definida da região cauda t para a região cabeça h. Por exemplo, na Figura 7.3 a seção definida por duas regiões adjacentes A e B é diferente da seção que liga B com A. Assim, uma seção é associada com sua seção oposta. A associação um-para-um Seção Oposta na Figura 7.2 mostra este relacionamento.

Existem três tipos de conectores de trilho na malha ferroviária: travessia, final de linha e desvio. Um conector travessia (Figura 7.4(d)) é um tipo estático de conector, que não muda de direção. Um conector final (Figura 7.4(c)) é um conector terminal na malha. Um conector de desvio é um conector que tem duas direções controláveis: reto e curvo.

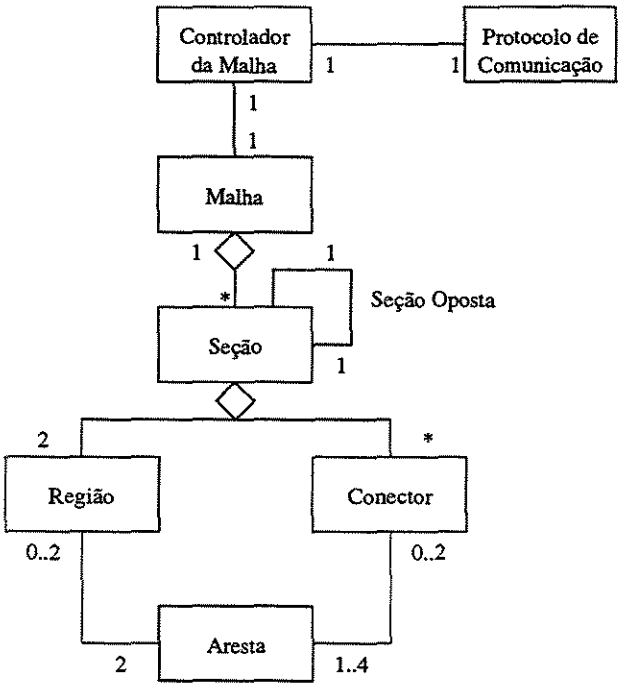


Figura 7.2: Malha ferroviária

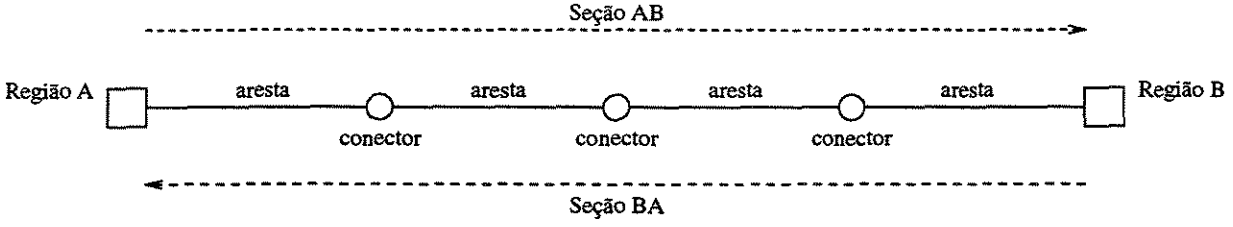


Figura 7.3: Exemplo de seção

Existem dois tipos de conectores de desvio: bifurcação (Figura 7.4(a)) e cruzamento (Figura 7.4(b)). Um conector bifurcação está associado a três arestas, enquanto um conector cruzamento está associado a quatro arestas. Um conector travessia está associado também a quatro arestas e um conector final de linha está associado a apenas uma aresta. A Figura 7.2 ilustra a associação entre as classes Conector e Aresta. Dependendo do tipo de conector, ele pode estar associado a uma ou até quatro arestas, de forma inversa, cada aresta pode estar associada a zero ou até dois conectores.

Dada uma seção S com uma região cauda t e uma região cabeça h, as próximas seções de S são as seções cuja estação cauda é h. Por exemplo, a Figura 7.5 ilustra uma parte da malha, a qual contém quatro regiões - região A, região B, região C e região D, um conector cruzamento nomeado cruzamento1, e dois conectores bifurcação nomeados bifurcação2 e bifurcação3. Assim, podemos identificar as seções descritas na tabela presente nesta figura. Por exemplo, as próximas seções da seção cuja região cabeça é a região A são seção 1, seção 3 e seção 5. As seções opostas de seção 1, seção 3 e seção 5 são respectivamente, seção 2, seção 4 e seção 6.

Nós podemos identificar três diferentes tipos de seções na malha: seção sólida, seção particionada e seção interconectada. Uma seção sólida tem próxima seção enquanto uma seção particionada não tem uma próxima seção. Uma seção interconectada se encontra na fronteira de duas partes da malha, ou seja, sua próxima seção é controlada por um Controlador da Malha remoto. Neste caso, o Controlador da Malha da parte em que começa a seção interconectada é responsável por comunicar-se com o Controlador da Malha da parte em que termina a seção, para estabelecer um protocolo de transferência de controle do trem entre as partes da malha.

Supondo que o comportamento desses três tipos de seções é estático, ou seja, conhecemos antecipadamente quais as seções que são sólidas, particionadas ou interconectadas, podemos criar uma hierarquia de classes derivadas (Sólida, Particionada e Interconectada) que herdam as características comuns de uma classe base Seção (Figura 7.6(a)). Os métodos Reserva e Libera implementam a funcionalidade básica de reserva e liberação de seções sólidas ou particionadas e são redefinidas na subclasse Interconectada. O método Ocupa implementa a funcionalidade básica de uma seção sólida e é redefinido na subclasses Particionada e Interconectada.

Conectores de trilho podem ser classificados em diferentes tipos: Final de Linha, Travessia e Desvio. O Tipo Desvio, por sua vez, pode ser classificado em diferentes tipos: Bifurcação e Cruzamento. A hierarquia de classes da Figura 7.6(b) encapsula os dados e comportamentos dos diversos tipos de conectores de trilho existentes na malha ferroviária (Figura 7.4).

A classe Conector define o atributo acesso, que indica se um conector está livre ou reservado, bem como as operações Reserva e Libera, que mudam o valor do atributo acesso.

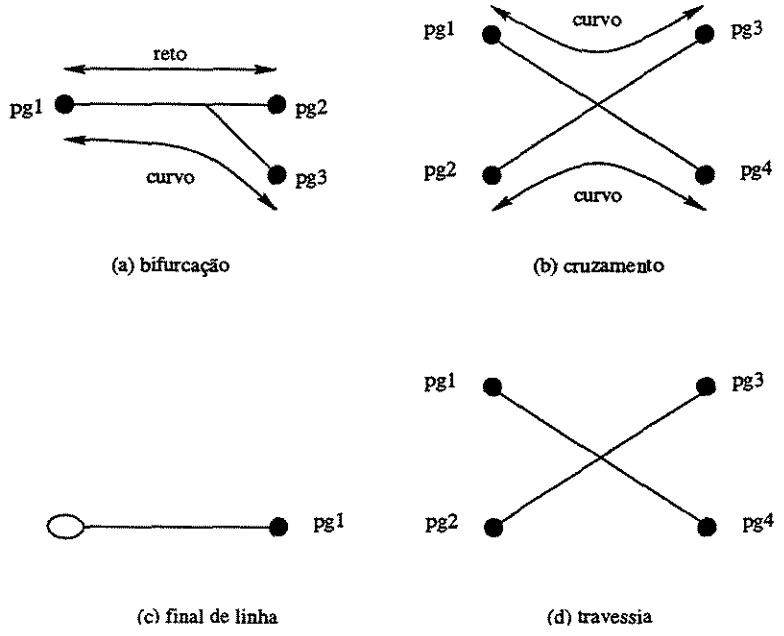
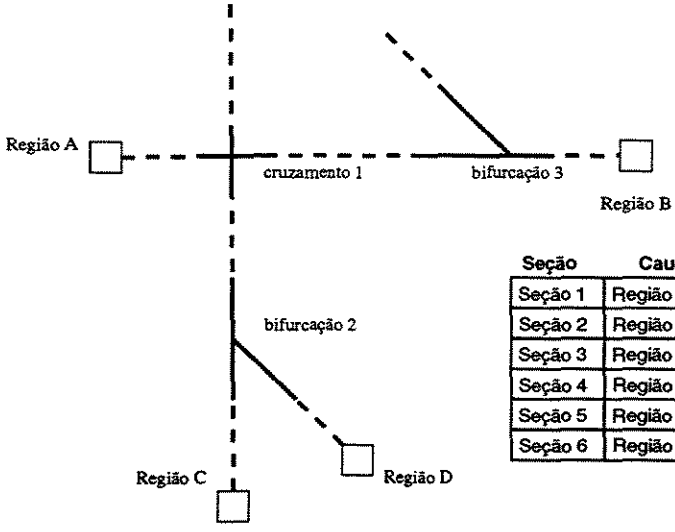
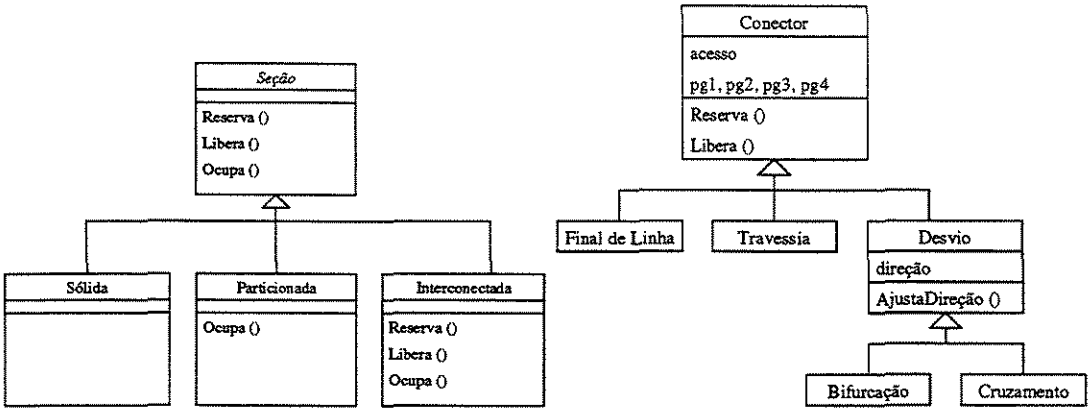


Figura 7.4: Tipos de conectores de trilho



Seção	Cauda	Cabeça	Caminho
Seção 1	Região A	Região B	cruzamento1 (reto), bifurcação3 (reto)
Seção 2	Região B	Região A	bifurcação3 (reto), cruzamento1 (reto)
Seção 3	Região A	Região D	cruzamento1 (curvo), bifurcação2 (curvo)
Seção 4	Região D	Região A	bifurcação2 (curvo), cruzamento1 (curvo)
Seção 5	Região A	Região C	cruzamento1 (curvo), bifurcação2 (reto)
Seção 6	Região C	Região A	bifurcação2 (reto), cruzamento1 (curvo)

Figura 7.5: Exemplo de próximas seções de uma seção



(a) Seção (b) Conector

Figura 7.6: Seções e conectores

Um conector contém também pontos de guia, definidos pelos atributos pg1, pg2, pg3 e pg4, que se conectam com as arestas. Dependendo do tipo de conector, o número de pontos de guia pode ser 1 (final de linha), 3 (bifurcação) ou 4 (travessia e cruzamento), como mostrado na Figura 7.4. A classe Desvio representa um tipo de conector que tem uma direção reta ou curva, como especificado pelo atributo direção, bem como uma operação AjustaDireção utilizada para mudar o valor do atributo direção.

7.3.2 Camada: controle de estações

O objetivo desta seção é apresentar o projeto detalhado da **camada de controle de estações**. Nós utilizamos o método discutido no Capítulo 6 no projeto desta camada. Esta seção descreve as atividades (e tarefas) deste método no contexto do projeto desta camada. As tarefas T6 e T10 não foram mencionadas nesta seção pois elas não foram utilizadas no projeto desta camada.

Atividade 1. Projetar os componentes e suas respostas excepcionais

Esta camada é constituída principalmente por dois tipos de componentes: estações e trens. As principais operações do componente Trem são: Inicia e Para um trem; Move, que executa o movimento do trem para uma nova seção; e Reverte, utilizada para reverter a direção do trem. É importante ressaltar que o trem deve se mover pela malha sem colisões. O conceito de região de controle é muito importante para atender este requisito. Uma Região de Controle é um conjunto de seções predefinidas (consecutivas) para o trajeto do

trem. Cada trem é responsável pela escolha de seu trajeto. Deste modo, um trem sabe qual é a próxima região do trajeto (ou seja, o próximo sensor a ser ativado).

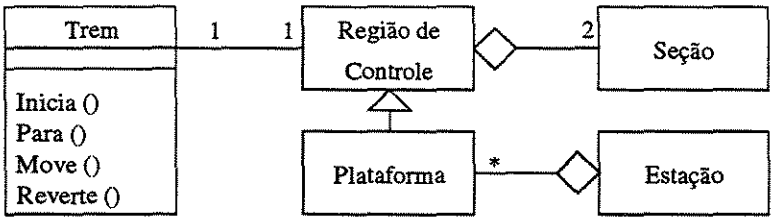


Figura 7.7: Trem, região de controle, plataforma e estação

Quando supomos que todos os conectores e sensores da malha são elementos confiáveis, é suficiente para um trem construir uma região de controle de apenas um nível. Além disso, não é necessário reservar todas as próximas seções da seção corrente; é suficiente reservar apenas uma próxima seção para garantir que não haverão colisões de trens. Entretanto, se considerarmos a possibilidade de ocorrerem falhas em sensores e conectores, o trem precisa reservar um número maior de seções. Desde que foi assumido que apenas uma falha pode acontecer nos sensores e conectores presentes em um trilho durante o intervalo de interesse (Seção 7.1.2), o trem precisa implementar uma Região de Controle com dois níveis (Figura 7.7). Uma Plataforma de uma Estação é um tipo especial de Região de Controle.

Tarefa T1. Utilizar contratos na especificação dos componentes. Nós utilizamos o padrão **contrato reflexivo** (Seção 5.1.1) para estruturar a classe TremContrato que realiza as atividades de detecção de erros. Esta classe (Figura 7.8) detecta os seguintes erros:

- Pós-condição da operação Para: se o próximo sensor da rota de um trem é ativado, então o trem falhou em executar a operação Para. Ou seja, o trem não conseguiu parar. Isto é detectado quando um diferente mas previsível sensor acusa a presença do trem.

- Pós-condição da operação Inicia: se o próximo sensor da rota de um trem não é ativado (detectado por *timeout*), então o trem falhou em executar a operação Inicia. Ou seja, o trem não conseguiu iniciar o seu movimento;

Tarefa T2. Separar as atividades normais e excepcionais dos componentes. Os padrões **estratégia de tratamento de exceções** (Seção 5.1.2) e **tratador** (Seção 5.1.3) permitiu-nos separar transparentemente as atividades normais e excepcionais do componente Trem

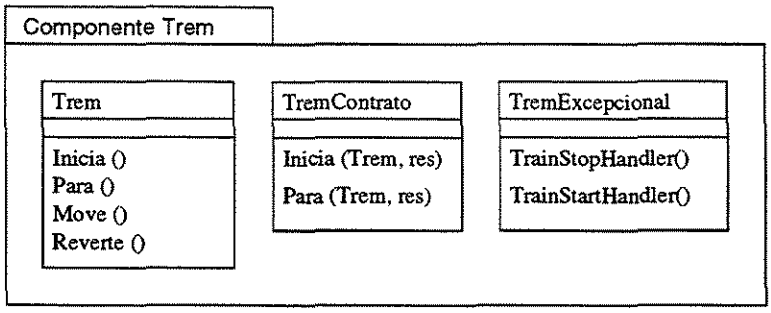


Figura 7.8: Projeto do componente Trem

(Figura 7.8). A classe TremExcepcional implementa os mecanismos de recuperação de erros para as situações em que o trem falha em iniciar ou parar seu movimento.

Tarefa T3. Representar exceções como classes. Nós utilizamos o padrão de projeto **exceção** para definir as exceções TrainStartException e TrainStopException (Figura 7.11) que correspondem as situações em que o trem falha em iniciar ou parar seu movimento.

Tarefa T4. Associar tratadores em multi-níveis. Neste estudo de caso, nós apenas utilizamos associação de tratadores a classes. Os métodos da classe TremExcepcional são os tratadores de classes para as exceções que deveriam ser tratadas no contexto dos métodos da classe Trem.

Tarefa T5. Utilizar propagação explícita de exceções e o modelo de terminação. Seguimos esta diretriz no projeto deste estudo de caso.

Atividade 2. Projetar as colaborações e os papéis de componentes

---

O componente Trem desempenha o papel de componente Conexão na sua interação com outros trens. O padrão de projeto **papel reflexivo** (Seção 5.3.1) foi utilizado no projeto deste papel de componente. O padrão **colaboração confiável** (Seção 5.4.1) foi usado para projetar as colaborações realizadas pelos trens. A classe Station estende a classe Collaboration, então instâncias desta classe representam as colaborações que coordenam a execução de uma atividade correspondendo a cooperação de dois trens chegando em uma estação particular ao mesmo tempo. As classes TrainEntry, Exchange e TrainExit estendem a classe Collaboration e correspondem as atividades aninhadas da atividade Station e que são responsáveis respectivamente por controlar a parada dos trens, troca de passageiros e a subseqüente saída de trens das estações ferroviárias. As classes TrainSensor e TrainActuator estendem Participant; instâncias destas classes representam os participantes da

colaboração (isto é, papéis do conector). Papéis de componentes são ativados por estes participantes. Instâncias de TrainActuator afetam a execução de instâncias de Trem (desempenhando o papel de componente Conexão) por enviar comandos para parar em estações e deixar estações após a atividade cooperativa ter terminado. Instâncias de TrainSensor estão associadas com instâncias de sensores que checam se trens não respondem a uma requisição. Figura 7.9 ilustra o projeto de uma atividade cooperativa entre dois trens parando na estação A (trem1 para na plataforma P1 e trem2 na plataforma P2). Plataformas P1 e P2 são modeladas como objetos associados à instâncias de MetaAtomic que garantem as suas semânticas transacionais.

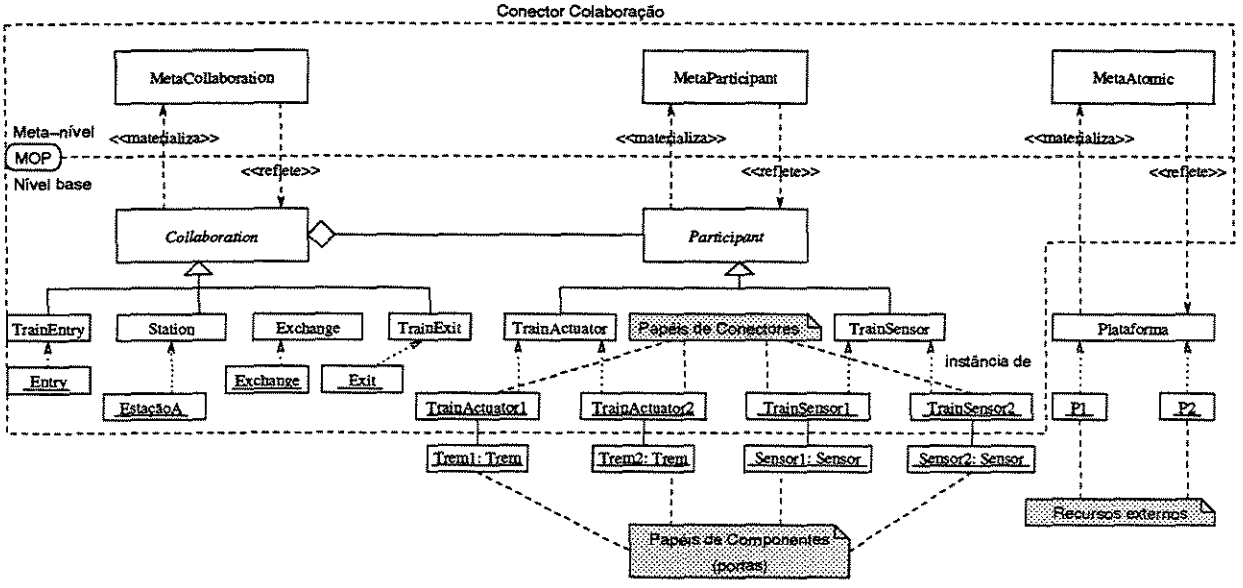


Figura 7.9: Estudo de caso: refinamento do conector colaboração

Atividade 3. Refinar os papéis de componentes

Tarefa T7. Utilizar contratos na especificação dos papéis de componentes. O padrão **contrato reflexivo** (Seção 5.1.1) foi utilizado para refinar as atividades de detecção de erros do componente Trem (Figura 7.10) e incluir os seguintes testes:

- Precondição da operação TroquePassageiros: ambos os trens devem chegar na estação antes de t_{Entry};
- Pós-condição da operação TroquePassageiros: ambos os trens devem deixar a estação até t_{Exit}.

Tarefa T8. Separar as atividades normais e excepcionais dos papéis de componentes. Os padrões **estratégia de tratamento de exceções** (Seção 5.1.2) e **tratador** (Seção 5.1.3) permitiu-nos separar transparentemente as atividades normais e excepcionais do componente Trem (Figura 7.10). Estas classes excepcionais implementam os tratadores para exceções locais e concorrentes levantados pelos participantes da atividade cooperativa.

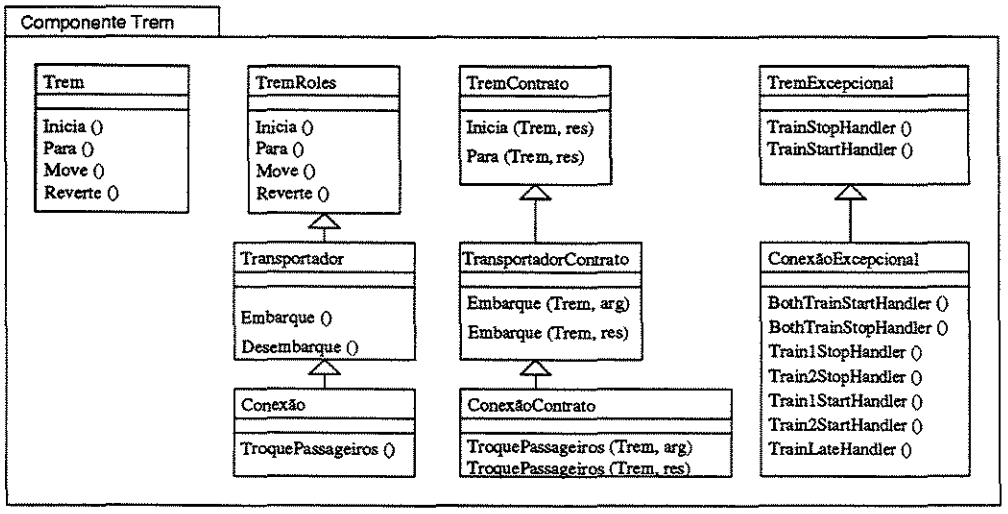


Figura 7.10: Refinamento do componente Trem

Lidando com falhas simples. Com propósitos de ilustração, nós apresentamos os requisitos básicos dos tratadores de três diferentes exceções simples:

- Se um trem (por exemplo, Trem1) não chega antes de t_{Entry}, então a Train1LateException é levantada pela colaboração aninhada (instância de Exchange). A colaboração (instância de Station) realiza a recuperação de erros para esta exceção em particular por embarcar todos os passageiros que devem embarcar no trem não falho, deixar alguns passageiros na estação esperando pelo trem atrasado e levantar a exceção TrainLateException que deveria ser tratada pela **camada de controle ferroviário** por rescalonar os passageiros do trem falho dado que estes podem ter perdido as suas conexões.

- Se um trem (por exemplo, Trem2) não consegue parar na estação (a operação Para não foi realizada com sucesso), então a colaboração aninhada (instância de TrainEntry) realiza a recuperação por invocar novamente a operação Para no trem e retornar a seção anterior da malha ferroviária. Se a falha persiste, então a exceção

Train2StopException é levantada. A colaboração (instância de Station) realiza a recuperação para esta exceção em particular por embarcar todos os passageiros que deveriam embarcar no trem não falho e produzir uma exceção (TrainStopException) que deveria ser tratada pela **camada de controle ferroviário** por rescalonar os passageiros que deveriam embarcar no trem falho e direcionar o trem para uma região segura da malha ferroviária, enquanto ao mesmo tempo parando os outros trens se existe possibilidade de colisão.

- Se um trem (por exemplo, Trem1) não consegue deixar a estação após a troca de passageiros ter sido realizada (a operação Inicia não foi realizada com sucesso), então a colaboração aninhada (instância de TrainExit) realiza a recuperação por invocar novamente a operação Inicia no trem. Se a falha persiste, então a exceção Train1StartException é levantada. A colaboração (instância de Station) realiza a recuperação de erros por tentar substituir o trem falho. Se um trem sobressalente não é disponível, esta colaboração desembarcará todos os passageiros deste trem e produzirá uma exceção TrainStartException que deveria ser tratada pela **camada de controle ferroviário** por rescalonar todos os passageiros do trem falho.

Lidando com falhas concorrentes. Vamos agora abordar o problema de possíveis falhas concorrentes (exceções estruturadas). Para este estudo de caso, dois tipos de falhas concorrentes foram identificadas. Com propósitos de ilustração, nós apresentamos os requisitos básicos dos tratadores destas duas diferentes exceções:

- A colaboração (instância de Station) não é hábil para realizar qualquer tipo de recuperação de erros se ambos trens não conseguem parar na estação (a operação Para em ambos trens não foi realizada com sucesso). A única coisa que esta colaboração pode fazer é levantar a exceção BothTrainStopException que deveria ser tratada pela **camada de controle ferroviário** por direcionar estes trens para uma região segura da malha ferroviária.

- Se ambos trens não conseguem deixar a estação, então a colaboração tenta realizar a recuperação de erros por tentar substituir os dois trens. Caso trens sobressalentes não estejam disponíveis, a colaboração desembarcará todos os passageiros de ambos trens e levantará a exceção BothTrainStartException que deveria ser tratada pela **camada de controle ferroviário** por rescalonar os passageiros que desembarcaram dos trens falhos.

Tarefa T9. Representar as exceções concorrentes como classes. Durante a execução de uma colaboração, se uma falha (de um componente envolvido nesta colaboração) ocorre e um erro é detectado, uma correspondente exceção será levantada por um dos participantes da colaboração. A exceção é propagada imediatamente para os outros participantes da colaboração e todos participantes então transferem o fluxo de execução para os respectivos tratadores desta exceção que tentam realizar a recuperação de erros apropriada. Nós usamos o padrão **exceção** (Seção 5.2.1) para definir as exceções simples e estruturadas uniformemente (Figura 7.11).

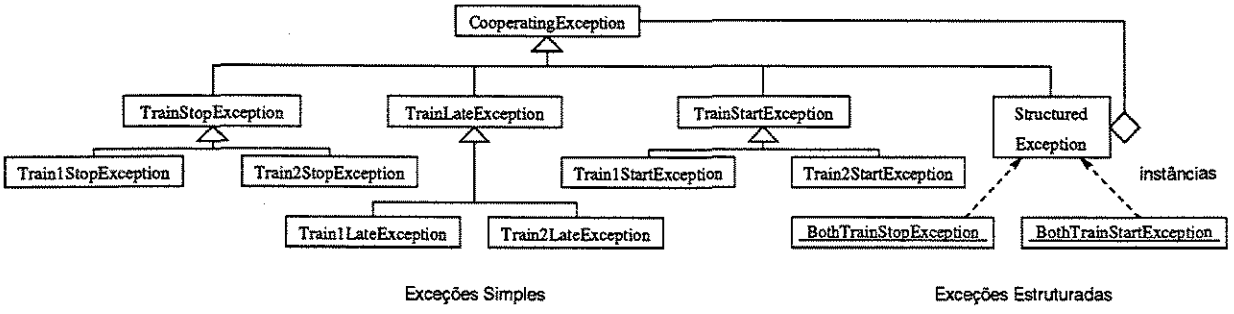


Figura 7.11: Estudo de caso: árvore de exceções

Atividade 4. Separar as propriedades funcionais das não-funcionais de seu sistema

Nós utilizamos o *framework* orientado a objetos (Seção 5.5) na codificação desta camada. Este *framework* permitiu-nos separar explicitamente as propriedades funcionais e não-funcionais de nossos componentes. Os arquivos de configuração utilizados na inicialização da meta-configuração associada a este estudo de caso são descritos no Apêndice A.

Utilizando tal *framework* permitiu-nos concentrar na definição da funcionalidade de nossa aplicação (por exemplo, as classes Station, TrainParticipant e Platform presentes na Figura 7.9) e abstrair da implementação das propriedades não-funcionais. No contexto deste estudo de caso estas propriedades seriam: a recuperação de erros coordenada, a sincronização dos participantes da colaboração e a atomicidade do acesso aos recursos externos à colaboração (instâncias da classe Platform).

7.3.3 Camada: controle ferroviário

O objetivo desta seção é apresentar o projeto detalhado da **camada de controle ferroviário**. Esta camada é constituída principalmente por três tipos de componentes: o controlador central, controladores de trens e o escalonador (Figura 7.12).

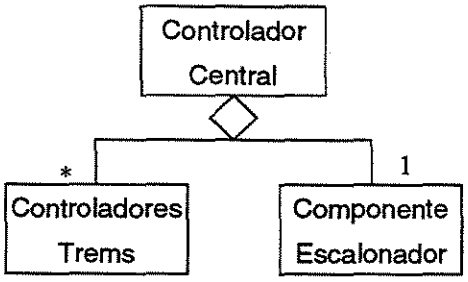


Figura 7.12: Camada controle ferroviário

Controladores de trens determinam os itinerários (e os respectivos horários) dos trens. Desta forma, controladores de trens determinam as estações ferroviárias em que trens param e em conseqüência a ordem em que as colaborações entre trens são executadas.

O componente escalonador determina o melhor (que leva o menor tempo) trajeto para os passageiros que desejam embarcar em uma estação origem O e desembarcar em uma estação destino D. É importante mencionar que uma diferente política poderia ter sido empregada, por exemplo, minimizar o número de conexões.

Nós modelamos a malha ferroviária como um grafo orientado com pesos nas arestas: vértices são estações e os pesos nas arestas representam a distância em minutos entre duas estações. Nós utilizamos uma versão modificada do algoritmo de Dijkstra [37, 29] que determina o caminho mais curto entre um vértice destino d a partir de um vértice de origem o que leva em conta o tempo de espera em uma estação em caso de conexões. Figura 7.13 ilustra duas situações onde este algoritmo apresenta dois distintos escalonamentos dependendo da configuração (horários dos trens) do sistema. Suponha que um passageiro deseja sair da estação A e desembarcar na estação F.

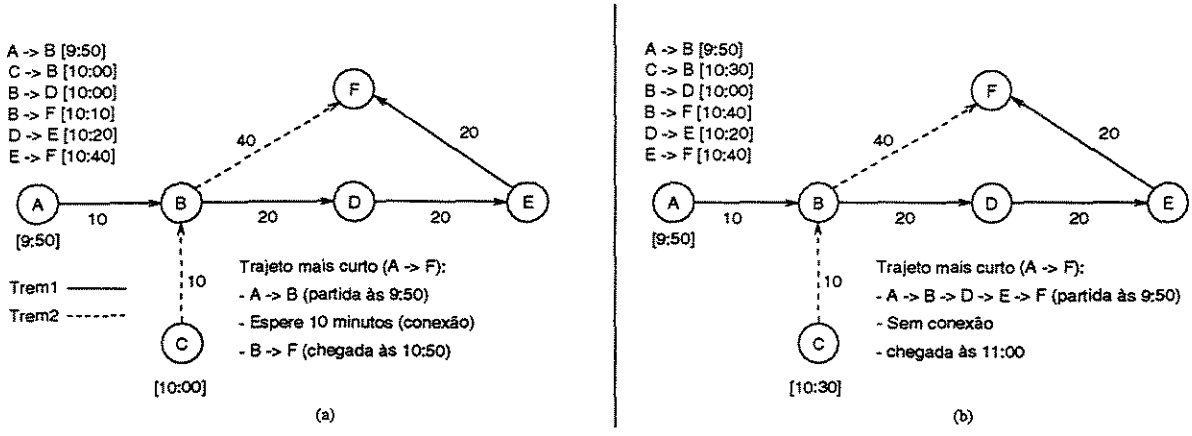


Figura 7.13: Estações e itinerários

- Na primeira situação (Figura 7.13(a)), o Trem1 está deixando a estação A às 9:50 enquanto que o Trem2 está deixando a estação C às 10:00. Nesta situação, o melhor escalonamento para sua viagem seria: embarcar no Trem1 às 9:50, desembarcar do Trem1 na estação B às 10:00, esperar 10 minutos, embarcar no Trem2 às 10:10 e desembarcar na estação F (seu destino) às 10:50.

- Na segunda situação (Figura 7.13(b)), o Trem1 está deixando a estação A às 9:50 enquanto que o Trem2 está deixando a estação C às 10:30. Nesta situação, o melhor escalonamento para sua viagem seria: embarcar no Trem1 às 9:50 e desembarcar na estação F (seu destino) às 11:00 sem fazer nenhuma conexão.

As principais operações do componente Controlador Central são: Inicia que inicia o movimento de todos os trens do sistema, Para que para o movimento de todos os trens, AdicionaTrem que adiciona um novo trem (e seu respectivo controlador) no sistema e AdicionaPassageiro que adiciona um passageiro (e o respectivo escalonamento para sua viagem) no sistema. O componente Controlador Central é responsável pelo tratamento das exceções levantadas pela **camada de controle de estações ferroviárias**. Logo, nós refinamos o projeto deste componente para incluir o tratadores para estas exceções (Figura 7.14).

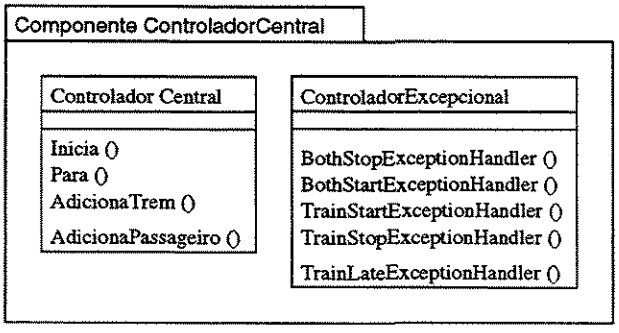


Figura 7.14: Componente controlador central

7.4 Considerações finais

Este estudo de caso foi implementado utilizando o *framework* reflexivo discutido na Seção 5.5 que provê uma infra-estrutura genérica para o desenvolvimento de sistemas concorrentes confiáveis e uma biblioteca de classes em Java [114] disponível na Universidade de Newcastle upon Tyne, Inglaterra que provê ferramentas para executar operações

genéricas para o controle de uma malha ferroviária[4].

Nossa análise, apresentada em Beder *et al.* [12, 11], mostra que este estudo de caso pode ser estendido em diversas direções proporcionando uma família de estudos de caso que nos permite aplicar os estilos arquiteturais e os respectivos padrões de projeto em um contexto mais amplo. Alguns exemplos de extensões são: considerar diferentes tipos de trens e de itens a serem transportados (cartas, mercadorias, etc) e relaxar as suposições sobre o comportamento do sistema descritas na Seção 7.1.2.

Outra maneira de estender este estudo de caso é introduzir sistemas-componentes autônomos que são controlados separadamente mas que são conectados através da malha ferroviária: cidades e aeroportos são conectados pelo sistema ferroviário através de estações. O desafio para pesquisa futura é desenvolver um sistema que provê suporte às funcionalidades complexas da integração de tais sistemas-componentes (vide discussão sobre sistemas-de-sistemas no Capítulo 9).

[4] A implementação deste estudo de caso foi realizada durante o período em que o aluno esteve visitando a Universidade de Newcastle upon Tyne, Inglaterra (Novembro/99 - Agosto/00).

Capítulo 8

Outros Exemplos de Uso da Arquitetura Proposta

Este capítulo apresenta dois exemplos do uso da arquitetura com o objetivo de salientar que a abordagem proposta por este trabalho é aplicável ao desenvolvimento de um grande leque de aplicações com características distintas. Seção 8.1 apresenta um estudo de caso onde diversidade de projeto é utilizada e onde a política de recuperação empregada é apenas utilizar a recuperação de erros por retrocesso. Seção 8.2 apresenta um estudo de caso onde os participantes da colaboração são mutuamente desconfiados (e portanto aspectos de segurança são essenciais) e que recursos externos não são acessados pelos participantes da atividade cooperativa.

8.1 Uma aplicação bancária

Este estudo de caso é baseado no apresentado por Vachon *et. al* [126]. A agência bancária deste estudo de caso oferece aos seus clientes um tipo de conta bancária chamada conta conjunta. Uma conta conjunta é composta por duas contas simples e pertence a dois clientes. Uma conta simples tem um estado interno que registra o saldo corrente da conta e um conjunto de operações.

Cada conta simples possui um cliente titular que é responsável pelas principais operações realizadas nesta conta: por exemplo, João é responsável pelas operações em conta-A, enquanto Maria é responsável pelas operações em conta-B. Cada cliente titular possui um número de identificação pessoal (NIP) que ele usa para se identificar.

Para uma conta conjunta um conjunto de condições deve ser definido pelos clientes e deve ser verificada quando uma operação é realizada. Por exemplo, se um dos titulares, João, deseja fazer um débito de R$ 100,00, uma atividade cooperativa, ao invés de uma

simples atividade, será realizada dado que a condição de que nenhum titular pode debitar qualquer valor da conta conjunta sem ter a autorização do outro titular é sempre assegurada (Figura 8.1). Os dois titulares da conta entram no contexto de uma colaboração (DébitoConjunto) por enviar ao outro titular o seu respectivo número de identificação pessoal. Após a atividade de checagem dos números de identificação pessoais de ambos titulares ter sido realizada, o valor é debitado no contexto de uma atividade cooperativa aninhada (Débito). Uma vez que exceções podem ocorrer durante a execução desta atividade aninhada, apenas a computação da atividade aninhada é perdida e o processo de autorização (checagem dos números de identificação pessoais) dos titulares não precisa ser refeita.

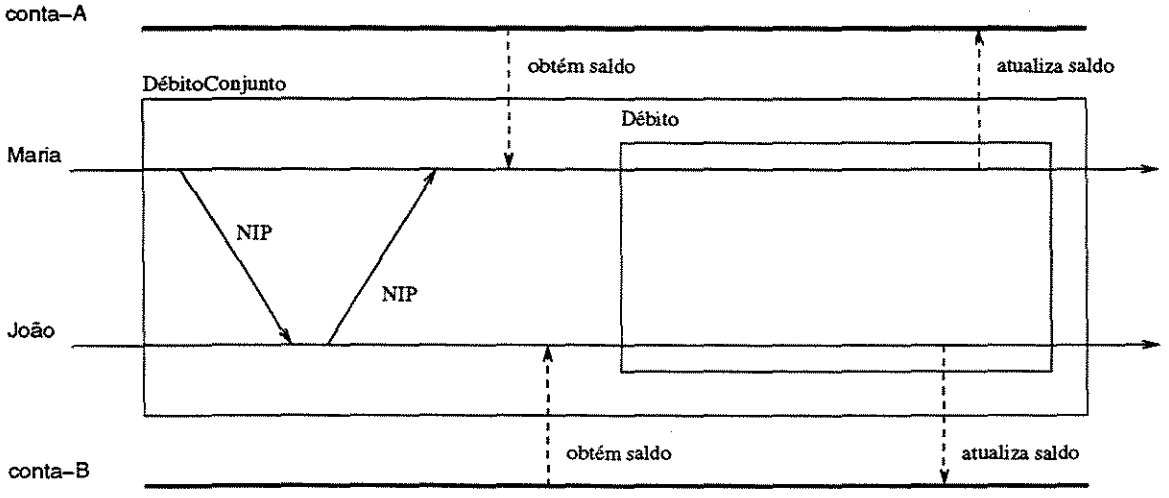


Figura 8.1: Atividade cooperativa DébitoConjunto

As condições aplicadas para contas conjuntas e NIPs são as seguintes:

- Número de identificação pessoal
  - Um cliente titular concorda com a execução de uma operação por fornecer seu NIP, que conseqüentemente necessita ser validado. Se o cliente titular enviar um NIP inválido a operação é imediatamente abortada.
- Contas conjuntas
  - Qualquer quantia pode ser depositada em qualquer conta simples (e conseqüentemente na conta conjunta) sem nenhuma autorização.
  - O saldo da conta conjunta (a soma dos saldos das contas simples) pode ser consultada por ambos os titulares.

- Uma operação de débito requer a autorização de ambos os titulares. Após passar pela atividade que checa os NIPs de ambos os titulares, o débito segue a seguinte política:
  1. Tenta-se debitar 100% do valor solicitado da conta do Sacador. Nesta atividade, o titular que solicita o débito é chamado de Sacador enquanto que o outro é chamado de Parceiro.
  2. Se o débito não é possível pois não existe dinheiro suficiente na conta do Sacador, tenta-se debitar 80% do valor solicitado da conta do Sacador e 20% do valor solicitado da conta do Parceiro.
  3. Se o débito ainda não foi possível, tenta-se debitar 60% do valor solicitado da conta do Sacador e 40% da conta do Parceiro.
  4. Como última alternativa, tenta-se retirar todo o saldo da conta do Sacador (isto é, zerar a sua conta) e retirar o restante (o que falta para completar o valor solicitado) da conta do Parceiro.
  5. Caso a última alternativa ainda não seja possível, então significa que este débito não pode ser realizado pois o saldo da conta conjunta é menor que o valor solicitado. Neste caso, uma exceção será levantada e a operação será abortada.
- Como mencionado, uma operação de débito em uma conta conjunta requer autorização de ambos os clientes titulares. Conseqüentemente, o titular precisa não apenas prover o seu NIP mas também solicitar o NIP do outro titular. Note que titulares tem apenas uma única oportunidade de proverem os seus NIPs quando são solicitados. Se um NIP inválido é fornecido, então a exceção InvalidPINException será levantada, e a operação será abortada.

8.1.1 Projeto detalhado

O objetivo desta seção é apresentar o projeto detalhado deste estudo de caso. Nós seguimos as diretrizes discutidas no Capítulo 6 na definição do projeto deste estudo de caso. Esta seção descreve as atividades (e tarefas) deste método no contexto do projeto deste estudo de caso. As tarefas T2, T4, T6, T7 e T8 não foram mencionadas nesta seção pois elas não foram utilizadas no projeto deste estudo de caso.

Atividade 1. Projetar os componentes e suas respostas excepcionais

Este estudo de caso apresenta principalmente dois tipos de componentes: clientes da agência bancária (Figura 8.3) e a conta conjunta formada por duas contas simples. Uma

conta simples tem um estado interno que registra o saldo corrente da conta e um conjunto de operações, tais como (i) Depositar() que deposita uma certa quantia na conta; (ii) Debitar() que debita uma certa quantia da conta; (iii) retornarSaldo() que retorna o saldo corrente da conta e verificarSaldo() que recebe um valor como parâmetro e retorna true se o saldo da conta corrente é maior ou igual ao valor do parâmetro e false, caso contrário.

Tarefa T1. Utilizar contratos na especificação dos componentes. Nós utilizamos o padrão **contrato reflexivo** (Seção 5.1.1) para estruturar a classe ContaContrato que realiza as atividades de detecção de erros (Figura 8.2). Nesta figura, apenas ilustramos as precondições e pós-condições associadas ao método Debitar. Os testes de precondições invocam a operação checarSaldo que retorna true se o saldo atual é maior ou igual ao valor solicitado a ser debitado e false, caso contrário. Se o teste de precondição não é satisfeito, a exceção InsufficientBalanceException é levantada. O teste de pós-condições e o de invariantes checam se após o débito, o saldo não se tornou negativo.

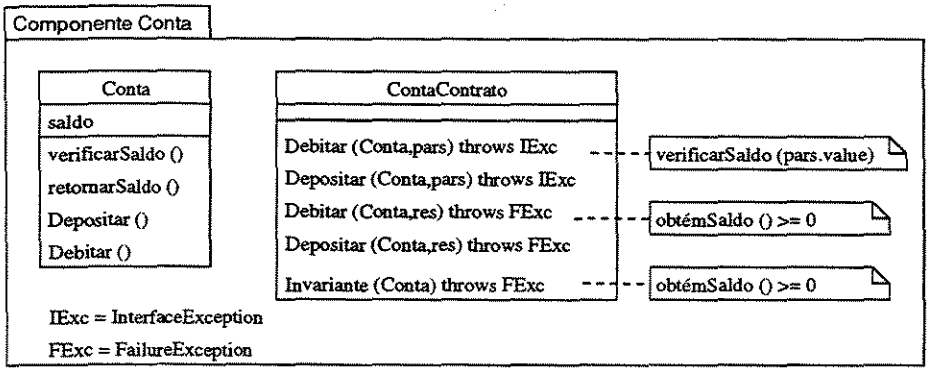


Figura 8.2: Especificação do componente Conta Bancária

Tarefa T3. Representar exceções como classes. Nós utilizamos o padrão de projeto **exceção** (Seção 5.2.1) para definir a exceção InsufficientBalanceException (Figura 8.5).

Tarefa T5. Utilizar propagação explícita de exceções e o modelo de terminação. Seguimos esta diretriz no projeto deste estudo de caso.

Atividade 2. Projetar as colaborações e os papéis de componentes

---

Este estudo de caso apresenta uma atividade cooperativa chamada DébitoConjunto desempenhada pelos dois clientes titulares de uma conta conjunta. É uma atividade cooperativa devido a condição de que nenhum cliente pode debitar nenhum valor sem a autorização do outro cliente.

Uma pessoa pode desempenhar diferentes papéis nas suas interações em diferentes contextos. Figura 8.3 ilustra o uso do padrão **papel reflexivo** (Seção 5.3.1) no projeto de alguns destes papéis do componente Pessoa. João e Maria são duas instâncias deste componente. Nós modelamos titulares de uma conta bancária como um dos papéis de componentes que este componente pode desempenhar. Este papel possui um atributo que representa o número de identificação pessoal (NIP) e dois métodos: (i) getNIP que retorna o NIP; (ii) Débito que implementa a atividade cooperativa do débito em uma conta conjunta. As subclasses Sacador e Parceiro redefinem o método Débito e representam os dois papéis que os clientes podem desempenhar na colaboração DébitoConjunto.

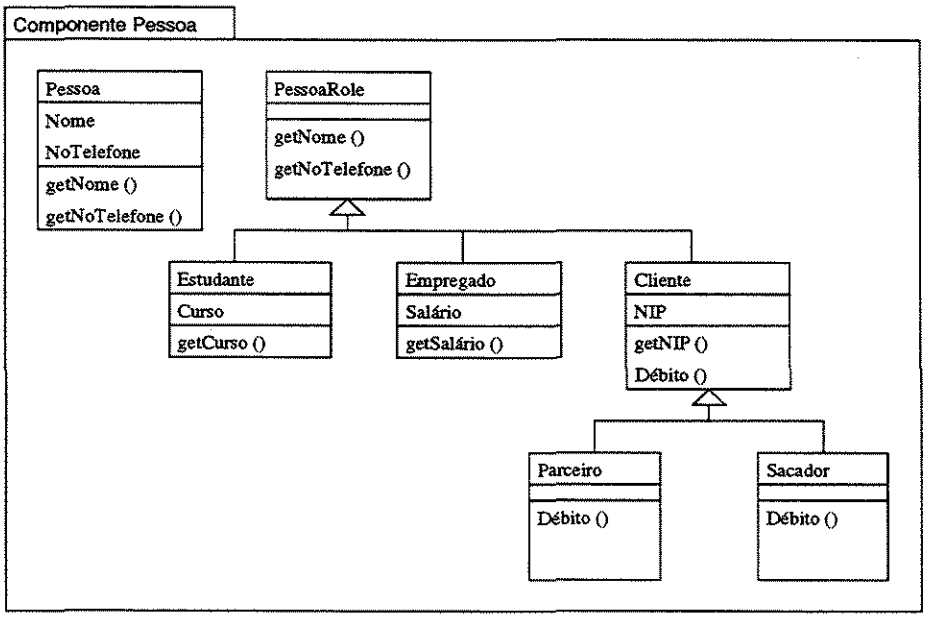


Figura 8.3: Componente Pessoa

O padrão **colaboração confiável** (Seção 5.4.1) foi utilizado para projetar as colaborações discutidas anteriormente. As classes DébitoConjunto e Débito estendem a classe Collaboration e instâncias destas classes representam as colaborações que coordenam a execução da operação de débito em uma conta conjunta. A classe ClientParticipant estende Participant; instâncias desta classe representam os participantes da colaboração (isto é, papéis do conector). Papéis de componentes são ativados por estes participantes. Instâncias de ClientParticipant afetam a execução de instâncias de Pessoa (Figura 8.3) desempenhando os papéis de **Sacador** ou **Parceiro** por enviar comandos para realizar o débito em uma conta conjunta. Figura 8.4 ilustra o projeto da atividade cooperativa entre João e Maria responsável pelo débito de uma certa quantia em uma conta conjunta. Nesta figura, Maria desempenha o papel de Sacador enquanto João desempenha o papel de Parceiro.

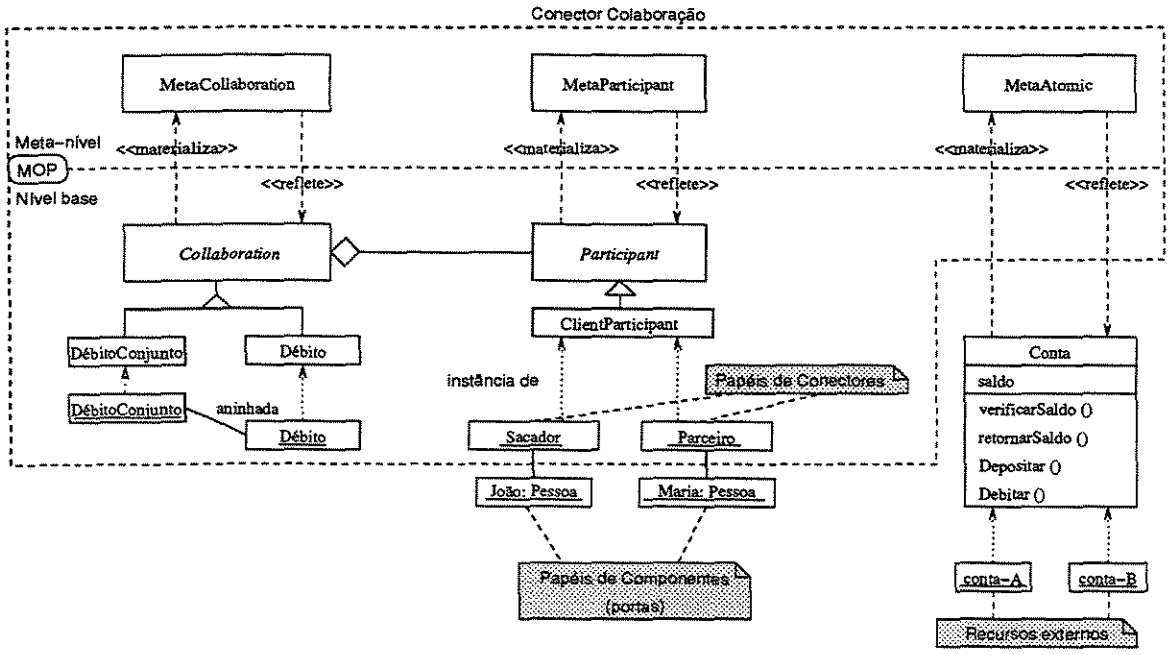


Figura 8.4: Projeto detalhado da colaboração DébitoConjunto

Atividade 3. Refinar os papéis de componentes

Tarefa T9. Representar as exceções concorrentes como classes. Nós utilizamos o padrão **exceção** (Seção 5.2.1) para estruturar as exceções que podem ser levantadas durante a execução da atividade cooperativa. A Figura 8.5 ilustra a hierarquia de exceções associada a este estudo de caso. A exceção InsufficientBalanceException estende InterfaceException e é levantada pelas contas bancárias (teste de precondições do método Debitar). As exceções NotEnoughMoneyException e InvalidPINException estendem CooperatingException e correspondem às exceções simples. Para as duas exceções, o mecanismo de recuperação de erros por retrocesso é empregado.

Tarefa T10. Utilizar diversidade no projeto dos papéis de componentes. Como discutido, a operação Débito (a atividade aninhada) adota a seguinte política:

- Tenta-se debitar 100% do valor solicitado da conta do **Sacador**. Nesta atividade, o titular que solicita o débito é chamado de **Sacador** enquanto que o outro é chamado de **Parceiro**.
- Se o débito não é possível pois não existe dinheiro suficiente na conta do **Sacador**, tenta-se debitar 80% do valor solicitado da conta do **Sacador** e 20% do valor solicitado da conta do **Parceiro**.

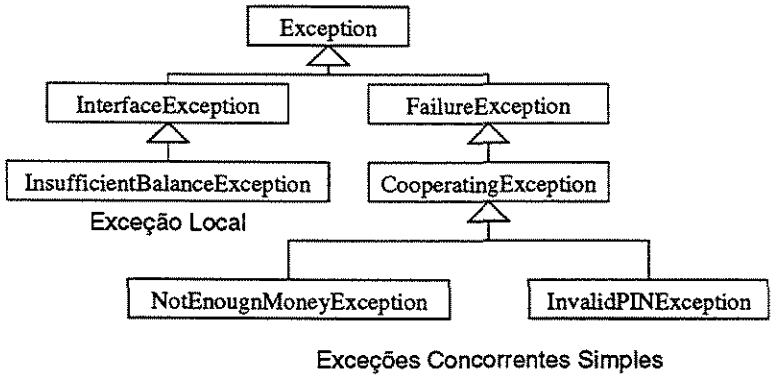


Figura 8.5: Exceções locais e concorrentes

- se o débito ainda não foi possível, tenta-se debitar 60% do valor solicitado da conta do **Sacador** e 40% da conta do **Parceiro**.
- como última alternativa, tenta-se retirar todo o saldo da conta do **Sacador** e retirar o restante (o que falta para completar o valor solicitado) da conta do **Parceiro**.
- Se a operação não pode ser realizada com sucesso, então a exceção NotEnoughMoneyException deve ser levantada.

Figura 8.6 ilustra os métodos dos papéis **Sacador** e **Parceiro** sendo implementados por diferentes projetos redundantes objetivando seguir a política descrita anteriormente (a técnica de bloco de recuperação é utilizada para este fim). Os participantes da atividade cooperativa tentam executar a primeira alternativa (100% para o **Sacador** e 0% para o **Parceiro**). Se a exceção InsufficientBalanceException é levantada por alguma conta bancária, então o titular desta conta trata esta exceção por levantar a exceção NotEnoughMoneyException. O tratamento desta exceção consiste em restaurar os estados dos objetos e tentar uma nova alternativa. Se nenhuma das alternativas é executada com sucesso, então a recuperação por retrocesso é utilizada novamente e os estados das contas bancárias são restaurados ao estado anterior ao início da colaboração DébitoConjunto.

Atividade 4. Separar as propriedades funcionais das não-funcionais de seu sistema

Nós utilizamos o *framework* orientado a objetos (Seção 5.5) na codificação deste estudo de caso. Este *framework* permitiu-nos separar explicitamente as propriedades funcionais e não-funcionais de nossos componentes. Os arquivos de configuração utilizados na inicialização da meta-configuração associada a este estudo de caso são descritos no Apêndice A.

Utilizando tal *framework* permitiu-nos concentrar na definição da funcionalidade de nossa aplicação (por exemplo, as classes Débito, DébitoConjunto, ClientParticipant e Conta

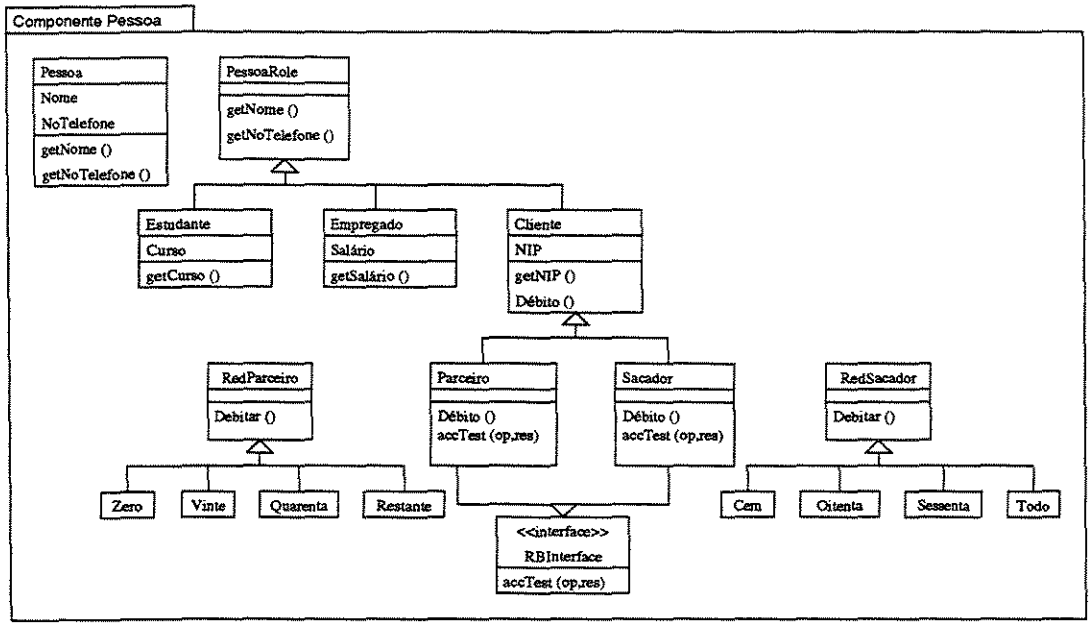


Figura 8.6: Parceiros e sacadores redundantes

presentes na Figura 8.4) e abstrair da implementação das propriedades não-funcionais. No contexto deste exemplo estas propriedades seriam: a recuperação de erros por retrocesso, a sincronização dos participantes da colaboração, a atomicidade do acesso aos recursos externos à colaboração (instâncias da classe Conta) e a diversidade no projeto dos papéis de componentes (instâncias das classes Parceiro e Sacador).

8.2 Implementação de um protocolo de justiça forte

Considere o seguinte cenário: Alice deseja comprar uma passagem aérea para uma viagem que fará nas suas férias. Existem diversos sítios na internet que vendem passagem aéreas. Entre estas, Alice decide comprar através do sítio da empresa BobAir, que oferece um preço reduzido desde que o pagamento seja feito antes de uma data especificada. O processo de comprar uma passagem aérea consiste em dois passos distintos: reserva e pagamento/entrega da passagem. Alice envia primeiro uma requisição de reserva. BobAir reserva um lugar e envia uma confirmação para Alice.

Entretanto, Alice e BobAir não necessariamente confiam um no outro. Considere os possíveis problemas decorrentes desta mútua desconfiança. Após fazer a reserva, Alice pode não retornar para comprar a passagem. BobAir é prejudicado financeiramente dado que o lugar reservado permanece vazio. Por outro lado, após reservar o lugar para Alice, BobAir pode vender este lugar para um outro passageiro que paga o preço integral da

passagem (sem desconto promocional), deixando Alice sem assento. Em cada caso, um dos lados sofre uma desvantagem apesar de seguir o protocolo corretamente. Nós dizemos que este protocolo não é justo [3].

A requisição de reserva feita pela Alice e a confirmação da reserva retornada por BobAir podem ser feitas não-repudiáveis por usar, por exemplo, técnicas de assinaturas digitais [87]. Desta forma, tendo enviado uma requisição de reserva, Alice não pode negar que fez a requisição. Similarmente, tendo uma confirmação de reserva, BobAir não pode negar que reservou uma assento para Alice. Mas isto não resolve completamente o problema. Suponha que Alice envia uma requisição não-repudiável. Se BobAir não retorna uma confirmação, Alice enfrenta um dilema. Se ela requisita uma reserva em um outro sítio, ela corre o risco de pagar duas passagens; se ela não toma nenhuma atitude, corre o risco de ficar sem passagem para sua viagem.

Quando um sistema envolve a participação de múltiplos, mutuamente desconfiados, participantes uma questão natural é se ele atende os requisitos de segurança de todos os participantes. Um sistema que não discrimina um participante que se comporta corretamente é dito ser justo. Se um participante se comporta corretamente, o sistema deve assegurar que os outros participantes não levam vantagem sobre este. O significado concreto de justiça depende de qual objetivo o protocolo deseja alcançar. A noção de justiça é bem estabelecido em processos reais tais como eleições e leilões.

Em caso de trocas, o significado de justiça é igualmente claro. Considere o caso do pagamento para uma mercadoria. Se o protocolo requer que Alice envie primeiro o pagamento, é um protocolo obviamente não justo.

Com o objetivo de vencer qualquer disputa subseqüente, Alice requer que ela seja garantida que ela vai receber a mercadoria desde que o sistema de pagamento eletrônico transferiu seu dinheiro para BobAir; ao mesmo tempo, BobAir requer que nenhuma mercadoria seja enviada para Alice ao menos que o dinheiro seja transferido. Esta é uma instância do problema genérico de troca justa. Em cenários de comércio eletrônico, você encontra outras instâncias de trocas onde justiça é um requisito crítico, por exemplo, assinatura de contratos eletrônicos. Em assinatura de contratos eletrônicos, cada participante troca um comprometimento não-repudiável para um texto de um contrato em retorno para um comprometimento de outro participante para o mesmo texto de um contrato.

Uma troca é justa se no fim da troca, ou cada participante recebe o item que espera ou nenhum participante recebe qualquer informação adicional sobre o item do outro participante. Isto é, a troca deveria ser atômica. Em cenários de comércio eletrônico, trocas podem ser realizadas utilizando uma rede de comunicação insegura. Um intruso pode obter controle sobre a rede ou corromper o sistema utilizado pelos participantes. Desta forma, protocolos de troca cuidadosamente projetados são necessários para garantir justiça.

Em um ambiente onde a maioria dos participantes comporta-se bem, nós podemos projetar protocolos eficientes por otimizar este caso comum. Nós confiaremos no uso de um terceiro participante[1], mas apenas nos casos excepcionais. A idéia básica é a seguinte. Primeiro, ambos participantes concordam nos itens a serem trocados e o que terceiro participante fará em casos excepcionais. Então, um dos participantes (chamado de originador) toma o risco de enviar o item primeiro, na esperança que o segundo participante (chamado de receptor) comporta-se-á corretamente e responderá com o seu item. Se o outro participante responde como esperado, o protocolo termina com sucesso. Caso contrário, o originador contata o terceiro participante para resolver a troca justa. Esta abordagem é conhecida como protocolo de **troca justa otimista**.

8.2.1 Protocolo otimista de assinatura de contratos

Nós ilustramos nesta seção um protocolo otimista de assinatura de contratos [3]. Em uma assinatura digital de contratos, ambos participantes já inicialmente concordaram no texto do contrato. Um contrato válido consiste de comprometimentos não-repudiáveis de cada participante no texto do contrato. Um protocolo de assinatura de contratos justo deveria assegurar que ambos participantes finalizam a execução do protocolo com contratos válidos ou nenhum deles.

Descrição do protocolo

Dois participantes O (originador) e R (receptor) desejam assinar um contrato digital de tal forma que ambos esperam finalizar com o comprometimento não-repudiável do outro participante sobre o texto de um contrato. Um terceiro participante T é conhecido de ambos os participantes. O protocolo de assinatura de contratos otimista [3] possui três sub-protocolos: troque, aborte e resolva. No caso normal, apenas o protocolo troque é executado. Os outros dois são usados apenas se um dos participantes decide forçar o término do protocolo (presumivelmente porque ele chegou a conclusão que algo está errado). Esta é uma escolha não-determinística feita localmente pelo participante. Note que, desde que o modelo é assíncrono, um participante chegando a conclusão que algo ocorreu errado não necessariamente implica que o outro participante comportou-se de forma errônea. O protocolo é ilustrado no Algoritmo 8.1. Lugares onde um participante pode decidir forçar o término do protocolo são representadas por **desiste?**. Por exemplo, **desiste?: aborte** significa que o participante decide desistir e executar o protocolo **aborte**.

[1] Em inglês: *third party*.

Nós assumimos que cada participante possui a capacidade de computar e verificar assinaturas digitais em algum esquema de assinatura digital em que:

- cada participante P tem uma chave de assinatura A_P, e uma correspondente chave de verificação V_P;
- cada participante P tem um algoritmo de assinatura assina tal que, dado uma mensagem m, assina$(m, A_P) = Ass_P(m)$, onde $Ass_P(m)$ é dito ser uma assinatura da mensagem m com a chave A_P; e
- cada participante P tem um algoritmo de verificação verifica tal que, dada uma mensagem m e uma assinatura s em m, verifica(m, s, V_P) retorna true se e somente se existe uma $Ass_P(m)$ que é igual a s.

Passo 1. É assumido que O e R já concordaram no texto do contrato e então executam o protocolo troque com o texto do contrato como entrada. O gera um número aleatório o_O e computa $com_O = h(o_O)$. com_O será usado como o comprometimento público para o segredo, O_o, significando que uma vez com_O é enviado para R, O não pode mudar o segredo o_O. $h()$ é uma função hash de única direção[2] que possui a seguinte propriedade: é computacionalmente impraticável determinar x, dado $h(x)$. O produz a mensagem $m_1 = V_O|V_R|T|texto|com_O$ e assina ela produzindo mt_1 que é enviada para R.

Observação: as mensagens são rotuladas de tal forma que a segunda letra do rótulo identifica o protocolo, e o subscrito identifica o número da mensagem. Por exemplo, mt_1 é a primeira mensagem do protocolo troque, e mr_2 é a segunda mensagem do protocolo resolve.

Passo 2. Se R decide desistir do protocolo, ele simplesmente abandona o protocolo. Em prática, R desistirá se não receber mt_1 dentro de um período de "tempo razoável". Note que não necessitamos relógios sincronizados: é responsabilidade de R decidir quanto tempo é este "tempo razoável".

Se m_1 não é formada corretamente, ou se verifica(m_1, mt_1, V_O) retorna false, R ignora a mensagem. Em prática, R pode enviar uma mensagem para O requisitando que este envie novamente m_1 e continuar esperando uma mensagem válida até o "tempo razoável" ter expirado. No restante da descrição, não explicitamente mencionaremos que mensagens mal formadas ou incorretas são ignoradas.

[2]Em inglês: *one-way hash function.*

Caso contrário, R gera um número aleatório o_R e computa $com_R = h(o_R)$. R gera a mensagem $m_2 = mt_1|com_R$ e assina ela produzindo mt_2 que é enviada para O.

Passo 3. Se O decide desistir, ele invoca T por executar o protocolo aborte. Em prática, O desistirá se não receber mt_2 dentro de um período de "tempo razoável". Caso contrário, ele envia o_O para R.

Passo 4. Se R decide desistir, ele invoca T por executar o protocolo resolva. Tipicamente, R desiste se ele não recebe um item x em tempo, de tal forma que $h(x) = com_O$. Caso contrário, ele envia o_R para O.

Passo 5. Se O decide desistir, ele invoca T por executar o protocolo resolva. Tipicamente, O desiste se ele não recebe um item y em tempo, de tal forma que $h(y) = com_R$.

O protocolo aborte é usado por O para abortar a troca de tal forma que T não irá resolver a troca no futuro. O protocolo resolve é usado por O ou por R para forçar um término com sucesso. Os protocolos resolve e aborte são garantidos serem atômicos.

No protocolo aborte, O envia uma requisição de aborto assinada para T. Se a correspondente troca não foi ainda resolvida, T envia um comprometimento do aborto, $Ass_T(abortado|mt_1)$. O comprometimento do aborto é uma garantia dado por T que ele ainda não resolveu e não resolverá esta instância particular de troque. Se o protocolo já foi resolvido, T simplesmente enviará o resultado do protocolo resolve.

No protocolo resolve, um dos participantes envia mt_1 e mt_2 para T. Se a troca já foi abortada, T simplesmente retorna o comprometimento do aborto. Caso contrário, ele envia um contrato substituto por assinar mt_1 e mt_2 (isto é, $Ass_T(mt_1|mt_2)$).

Um contrato válido é da forma

- $\{mt_1, o_O, mt_2, o_R\}$, ou $Ass_T(mt_1|mt_2)$

onde,

$mt_1 = Ass_O(V_O|V_R|T|texto|com_O)$,
$mt_2 = Ass_R(mt_1|com_R)$,
$com_O = h(o_O)$ e $com_R = h(o_R)$.

Algoritmo 8.1 Protocolo otimista de assinatura de contratos

Protocolo troque

O		R
$com_O = h(o_O)$	$\xrightarrow{mt_1 = Ass_O(V_O\|V_R\|T\|texto\|com_O)}$	**desiste?: abandone**
desiste?: aborte	$\xleftarrow{mt_2 = Ass_R(mt_1\|com_R)}$	$com_R = h(o_R)$
	$\xrightarrow{mt_3 = o_O}$	**desiste?: resolva**
desiste?: resolva	$\xleftarrow{mt_4 = o_R}$	

Protocolo aborte

O		T
	$\xrightarrow{ma_1 = Ass_O(abortado\|mt_1)}$	se resolvido **então**
		$ma_2 = Ass_T(mt_1\|mt_2)$
		senão abortado = verdade
		$ma_2 = Ass_T(abortado\|m$
	$\xleftarrow{ma_2}$	

Protocolo resolva

O,R		T
	$\xrightarrow{mr_1 = (mt_1, mt_2)}$	se abortado **então**
		$mr_2 = Ass_T(abortado\|m$
		senão resolvido = verdade
		$mr_2 = Ass_T(mt_1\|mt_2)$
	$\xleftarrow{mr_2}$	

8.2.2 Projeto do protocolo

O protocolo discutido na seção anterior supõe que as interações entre os participantes são totalmente assíncronas, que implica que a implementação dos protocolos aborte e resolva (em que T toma parte) torna-se um pouco complexa pois eles necessitam checar duas variáveis de estado (abortado e resolvido) e que estes protocolos são supostos serem atômicos.

Nós consideramos T como um componente crítico, logo nós argumentamos que é válido "relaxar" um pouco a assincronia da interação entre os participantes, objetivando tornar menos complexa os protocolos aborte e resolva e conseqüentemente a implementação do componente T. Desta forma, nós propomos um projeto para este protocolo com este fim ("relaxar" a assincronia da interação dos participantes visando facilitar a implementação do componente T).

O projeto proposto é baseado na observação que o protocolo troque é composto de duas fases distintas: troque comprometimentos (com_O e com_R) e troque segredos (o_O e o_R). Se os dois participantes estão ainda trocando comprometimentos, a assinatura do contrato ainda pode ser abortada. Se pelo menos um dos participantes já entrou na fase de troca de segredos, a assinatura do contrato não pode ser mais ser abortada, apenas resolvida. Nós argumentamos que se os participantes são sincronizados no início da fase troque segredos (isto é, apenas um participante inicia a execução desta fase, se ele está seguro que o outro participante possui mt_1 e mt_2) simplifica o projeto dos protocolos aborte e resolva.

Nós projetamos este protocolo como uma colaboração troque de três participantes (O, R e T) que possui duas colaborações aninhadas troque comprometimentos e troque segredos onde a segunda colaboração aninhada será apenas executada caso a primeira tenha terminado a sua execução com sucesso (Figura 8.7).

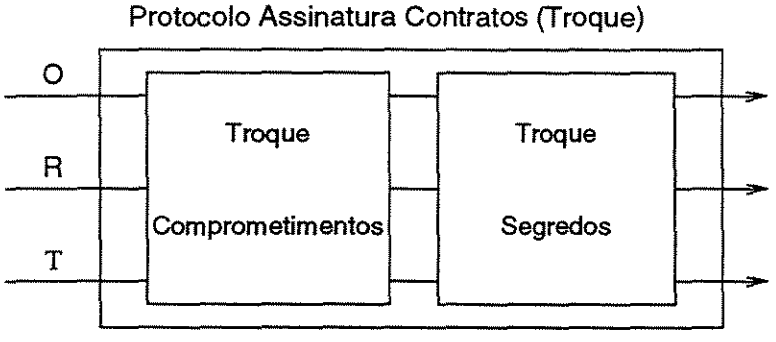


Figura 8.7: Projeto proposto do protocolo

Colaboração troque comprometimentos

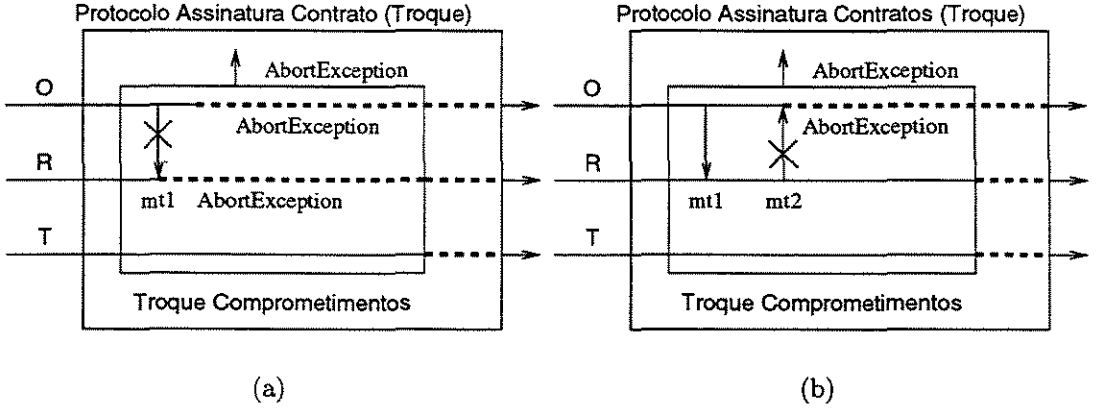
O		R
$com_O = h(o_O)$	$\xrightarrow{mt_1 = Ass_O(V_O\|V_R\|T\|texto\|com_O)}$	**desiste?: aborte**
desiste?: aborte	$\xleftarrow{mt_2 = Ass_R(mt_1\|com_R)}$	$com_R = h(o_R)$

Passo 1. O gera um número aleatório o_O e computa $com_O = h(o_O)$. O produz a mensagem $m_1 = V_O|V_R|T|texto|com_O$ e assina ela produzindo mt_1 que é enviada para R.

Passo 2. Se R decide desistir do protocolo, ele invoca T por executar o protocolo aborte. Caso contrário, R gera um número aleatório o_R e computa $com_R = h(o_R)$. R gera a mensagem $m_2 = mt_1|com_R$ que é enviada para O.

Passo 3. Se O decide desistir, ele invoca T por executar o protocolo aborte.

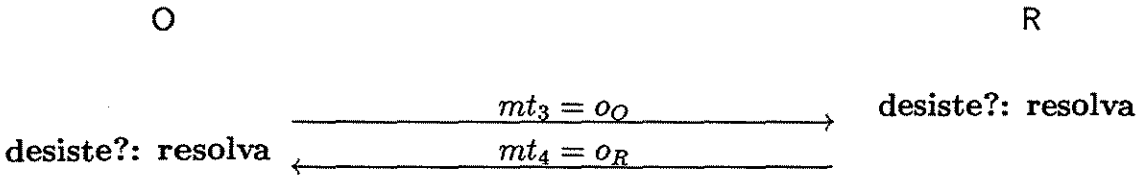
O protocolo aborte consiste em levantar a exceção AbortException que será tratada pelos três participantes por abortar a colaboração aninhada troque comprometimentos e sinalizar a mesma exceção para ser tratada no contexto da colaboração troque que consiste em abortar também esta colaboração. Figura 8.8 ilustra dois possíveis cenários onde a exceção AbortException é levantada. No primeiro cenário (Figura 8.8(a)), R não recebe mt_1 em tempo e portanto não envia mt_2 para O. Neste caso, os dois participantes levantam a exceção AbortException. No segundo cenário (Figura 8.8(b)), O não recebe mt_2 em tempo e portanto levanta a exceção AbortException.



(a) (b)

Figura 8.8: Cenários onde a exceção AbortException é levantada

Colaboração troque segredos



Passo 1. O envia o_O para R.

Passo 2. Se R decide desistir do protocolo, ele invoca T por executar o protocolo resolva. Caso contrário, R envia o_R para O.

Passo 3. Se O decide desistir, ele invoca T por executar o protocolo resolva.

O protocolo resolva consiste em levantar a exceção ResolveException que será tratada pelos três participantes por abortar a colaboração aninhada troque segredos e levantar a mesma exceção para ser tratada no contexto da colaboração troque. O tratamento desta exceção será realizado pelos três participantes e consiste em T assinar a mensagem $mt_1|mt_2$ (isto é, $Ass_T(mt_1|mt_2)$) e enviar para O e R. Figura 8.9 ilustra dois possíveis cenários onde a exceção ResolveException é levantada. No primeiro cenário (Figura 8.9(a)), R não recebe mt_3 em tempo e portanto não envia mt_4 para O. Neste caso, os dois participantes levantam a exceção ResolveException. No segundo cenário (Figura 8.9(b)), O não recebe mt_4 em tempo e portanto levanta a exceção ResolveException.

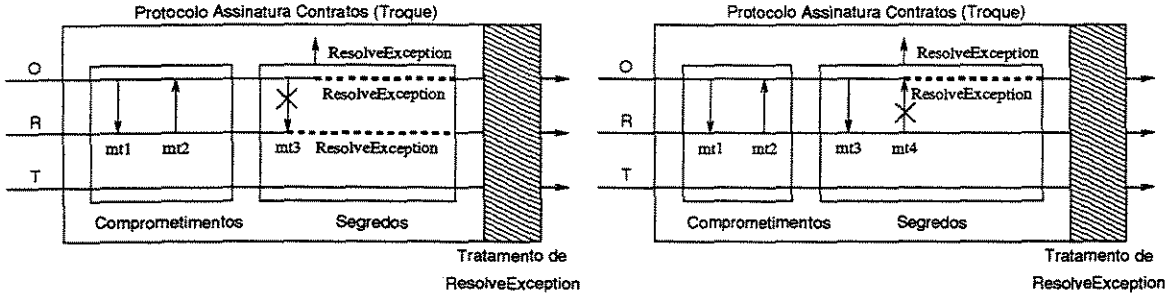


Figura 8.9: Cenários onde a exceção ResolveException é levantada

Projeto detalhado

O objetivo desta seção é ilustrar o projeto do protocolo discutido na seção anterior utilizando o método discutido no Capítulo 6. Esta seção descreve as atividades (e tarefas) deste método no contexto do projeto deste estudo de caso. As tarefas T1, T3, T6, T7 e T10 não foram mencionadas nesta seção pois elas não foram utilizadas no projeto deste estudo de caso.

Atividade 1. Projetar os componentes e suas respostas excepcionais

Este estudo de caso apresenta apenas um tipo de componente: os participantes do protocolo otimista de assinatura de contratos - Originador, Receptor e Terceiro (Figura 8.10).

Tarefa T2. Separar as atividades normais e excepcionais dos componentes. Os padrões **estratégia de tratamento de exceções** (Seção 5.1.2) e **tratador** (Seção 5.1.3) permitiu-nos separar transparentemente as atividades normais e excepcionais do componente Pessoa (Figura 8.12).

Tarefa T4. Associar tratadores em multi-níveis. Neste estudo de caso, nós apenas utilizamos associação de tratadores a classes. Os métodos da classe PessoaExcepcional (Figura 8.12) são os tratadores de classes para as exceções que deveriam ser tratadas no contexto dos métodos da classe Pessoa.

Tarefa T5. Utilizar propagação explícita de exceções e o modelo de terminação. Seguimos esta diretriz no projeto deste estudo de caso.

Atividade 2. Projetar as colaborações e os papéis de componentes

Este estudo de caso apresenta uma atividade cooperativa chamada troque desempenhada por três participantes (P, R e T). Como discutido anteriormente, uma pessoa pode desempenhar diferentes papéis de componentes em diferentes contextos. Figura 8.10 ilustra o uso do padrão **papel reflexivo** no projeto de alguns destes papéis do componente Pessoa. Nós modelamos o originador (O), o receptor (R) e o terceiro participante (T) como papéis do componente Pessoa. Estes três papéis estendem a classe abstrata *Signer&Verifier* que possui operações para assinar e verificar assinaturas digitais. Todas as três classes possuem nas suas interfaces o método assinaContrato que implementam as atividades desempenhadas por estes participantes no contexto da atividade cooperativa troque.

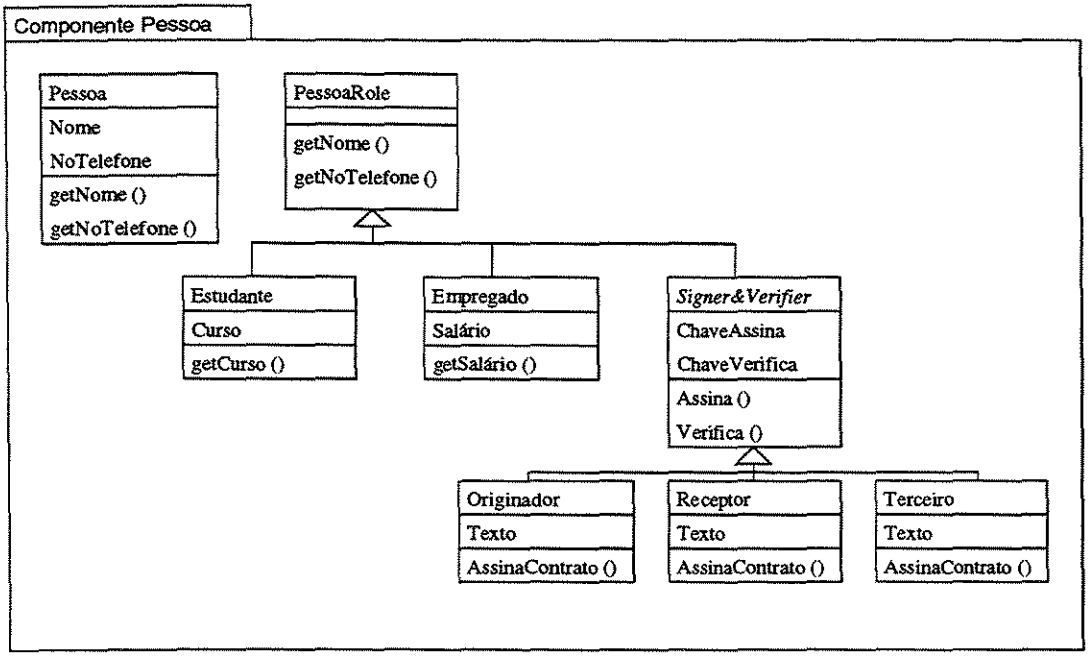


Figura 8.10: Componente Pessoa

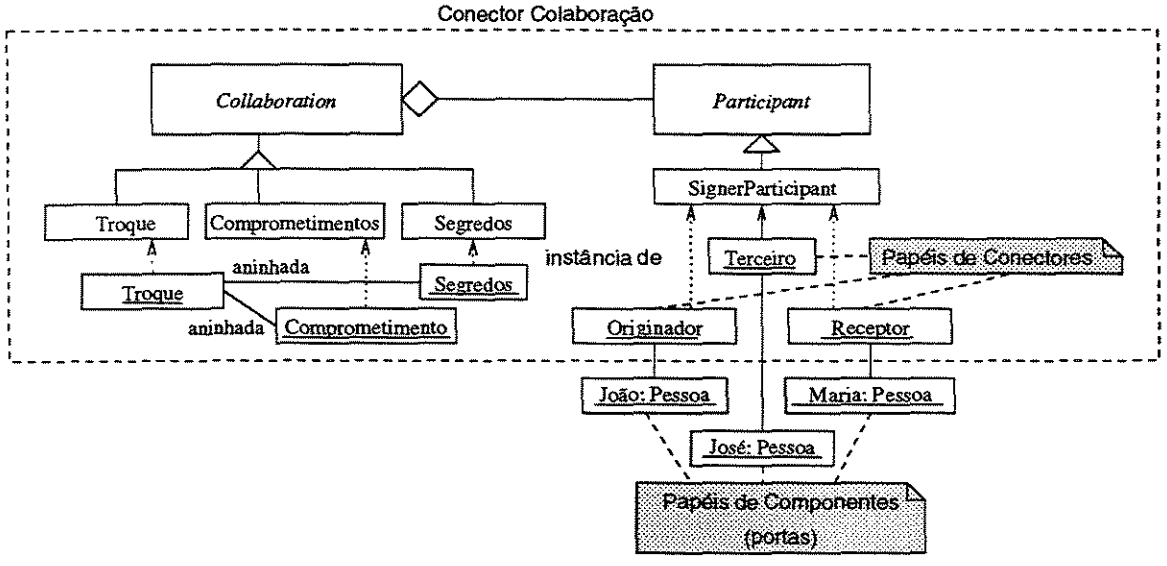


Figura 8.11: Projeto detalhado da colaboração Troque

O padrão **colaboração confiável** foi utilizada para projetar as colaborações discutidas anteriormente. As classes Troque, Comprometimentos e Segredos estendem a classe Collaboration e instâncias destas classes representam as colaborações que coordena a assinatura do contrato. A classe SignerParticipant estende Participant e instâncias desta classe representam os participantes da colaboração. Papéis de componentes são ativados por estes participantes. Instâncias de SignerParticipant afetam a execução de instâncias de Pessoa (Figura 8.10) desempenhando os papéis de **Originador**, **Receptor** ou **Terceiro**. Figura 8.11 ilustra o projeto da atividade cooperativa entre João, Maria e José responsável pela assinatura digital de um contrato. Nesta figura, João desempenha o papel de Originador enquanto Maria desempenha o papel de Receptor e José o papel de Terceiro.

Atividade 3. Refinar os papéis de componentes

Tarefa T8. Separar as atividades normais e excepcionais dos papéis de componentes. Os padrões **estratégia de tratamento de exceções** e **tratador** permitiu-nos separar transparentemente as atividades normais e excepcionais do papéis do componente Pessoa (Figura 8.12). Estas classes excepcionais implementam os tratadores para as exceções levantadas pelos participantes da atividade cooperativa.

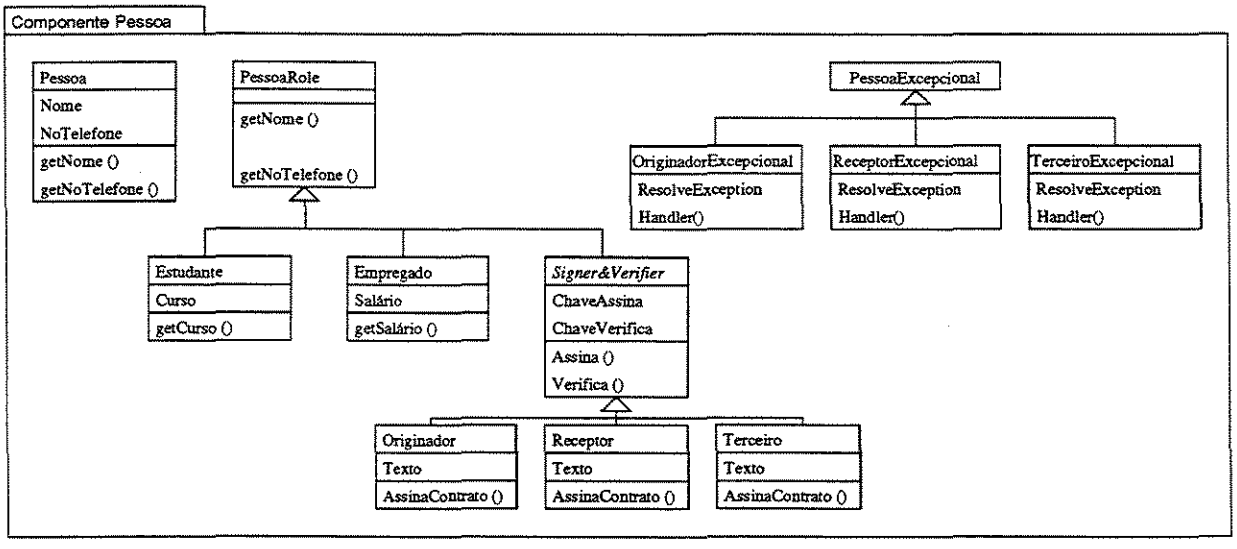


Figura 8.12: Refinamento componente Pessoa

Tarefa T9. Representar exceções concorrentes como classes. O padrão **exceção** foi utilizado para estruturar as exceções que podem ser levantadas durante a execução da atividade cooperativa. A Figura 8.13 ilustra a hierarquia de exceções associada com este estudo de

caso. As exceções AbortException e ResolveException são as exceções concorrentes simples que podem ser levantadas pelos participantes da atividade cooperativa.

- A exceção AbortException é levantada durante a execução da colaboração aninhada Comprometimentos e será tratada pelos três participantes por abortar esta colaboração aninhada e sinalizar a mesma exceção para ser tratada no contexto da colaboração troque que consiste em abortar também esta colaboração.
- A exceção ResolveException é levantada durante a execução da colaboração Segredos e será tratada pelos três participantes por abortar esta colaboração aninhada e levantar a mesma exceção para ser tratada no contexto da colaboração troque. O tratamento desta exceção no contexto da colaboração troque será realizado pelos três participantes e consiste em T assinar a mensagem $mt_1|mt_2$ (isto é, $Ass_T(mt_1|mt_2)$) e enviar para O e R.

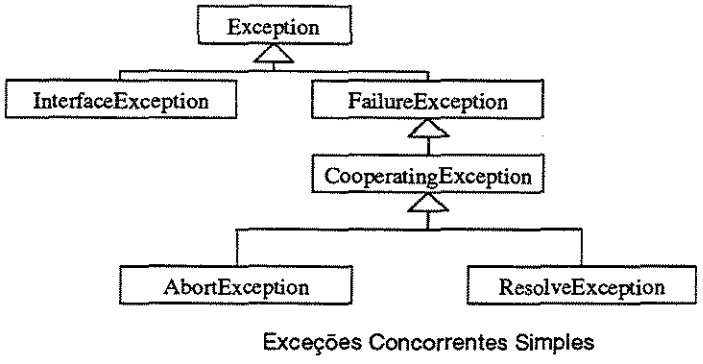


Figura 8.13: Exceções concorrentes

Atividade 4. Separar as propriedades funcionais das não-funcionais de seu sistema

Nós utilizamos o *framework* orientado a objetos (Seção 5.5) na codificação deste estudo de caso. Este *framework* permitiu-nos separar explicitamente as propriedades funcionais e não-funcionais de nossos componentes. Os arquivos de configuração utilizados na inicialização da meta-configuração associada a este estudo de caso são descritos no Apêndice A.

Utilizando tal *framework* permitiu-nos concentrar na definição da funcionalidade de nossa aplicação (por exemplo, as classes Troque, Comprometimentos, Segredos e SignerParticipant presentes na Figura 8.11) e abstrair da implementação das propriedades não-funcionais. No contexto deste exemplo estas propriedades seriam: a recuperação de erros coordenada e a sincronização dos participantes da colaboração.

8.3 Considerações finais

Os resultados desses experimentos (os três estudos de caso) geraram evidências que a arquitetura de software proposta proporciona uma maneira simples, flexível e efetiva para a construção de sistemas orientados a objetos confiáveis. Estes estudos de casos são baseados em aplicações complexas existentes no mundo real que foram simplificadas com propósitos de pesquisa. Isto é, a descrição destes estudos de casos foi tão realista quanto possível mas que ao mesmo tempo apresentou condições de mostrar a validade da pesquisa proposta nesta tese.

Capítulo 9

Conclusões e Trabalhos Futuros

Neste trabalho apresentamos uma arquitetura de software genérica para introduzir atomicidade, redundância de software, tratamento de exceções, detecção de erros e recuperação de erros coordenada no desenvolvimento de sistemas orientados a objetos confiáveis desde a fase de projeto arquitetural, passando pelo projeto detalhado e terminando na codificação/implementação do sistema.

Ademais, propusemos uma abordagem sistemática na incorporação de características de tolerância a falhas durante o ciclo de desenvolvimento do sistema, iniciando-se na fase de projeto arquitetural através do emprego de estilos arquiteturais, passando pelo projeto detalhado através do emprego de padrões de projetos que refinam os componentes lógicos e conectores da arquitetura de software proposta e terminando na fase de codificação através da reutilização de um *framework* orientado a objetos.

Finalmente, para verificarmos a aplicabilidade da arquitetura de software proposta, projetamos e implementamos três estudos de casos utilizando a abordagem proposta. Os resultados desses experimentos geraram evidências que nossa proposta proporciona uma maneira simples, flexível e efetiva para a construção de sistemas orientados a objetos confiáveis.

Durante o desenvolvimento deste trabalho, chegamos a vários resultados que formam as principais contribuições publicadas em revistas e conferências nacionais e internacionais:

- Definição de uma arquitetura de software genérica para o desenvolvimento de sistemas orientados a objetos confiáveis. A descrição da arquitetura de software proposta foi publicada em conferências nacionais [15] e internacionais [11].

- Definição de um conjunto coeso de padrões de projetos que refinam os elementos arquiteturais da arquitetura de software proposta e fornecem uma separação clara e transparente de interesses entre a funcionalidade da aplicação e a funcionalidade relacionada à provisão da confiabilidade do sistema. A descrição do conjunto de

padrões foi publicado em conferências nacionais [14] e internacionais [53] e em uma revista internacional [54].

- Definição de uma abordagem sistemática na incorporação de características de tolerância a falhas durante o ciclo de desenvolvimento do sistema. Esta abordagem identifica decisões de projeto adequadas para construir sistemas orientado a objetos confiáveis e apresenta um conjunto de diretrizes que deveria ser seguido durante o desenvolvimento do sistema.

- Implementação de 3 estudos de casos objetivando verificar a aplicabilidade da arquitetura de software proposta. A descrição do estudo de caso apresentado no Capítulo 7 foi publicado em uma conferência internacional [11] e em uma revista internacional [12].

Extensões

As principais linhas de pesquisa que podem ser seguidas a partir deste trabalho são:

- A descrição da arquitetura de software proposta apresentada neste trabalho foi realizada informalmente através da apresentação de diagramas informais que representam os componentes lógicos e conectores da arquitetura de software proposta. Uma possível extensão desse trabalho é utilizar uma linguagem de definição de arquiteturas, tais como Aesop [55], Rapide [80], Unicon [107] e Wright [2], na descrição formal dos elementos arquiteturais da arquitetura de software proposta. Linguagens de definição de arquiteturas[1] têm sido propostas no intuito de documentar formalmente as propriedades de um sistema. Linguagens de definição de arquiteturas endereçam a identificação e especificação dos componentes e conectores - elas tipicamente provêem suporte à especificação de ambas as características funcionais e não funcionais dos componentes e contem estruturas para especificar as propriedades de conectores que definem a interação entre componentes.

- A arquitetura de software proposta proporciona suporte à codificação de sistemas orientados a objetos confiáveis através da definição de um *framework* orientado a objetos. Projetistas reutilizam este *framework* de acordo com as características de suas aplicações. Outra maneira de aumentar a produtividade seria a reutilização de componentes de software (Seção 3.5) existentes. Tecnologias baseadas em componentes, tais como CORBA [91] e Enterprise Java Beans [121], permitem a construção de software através da reutilização de componentes de software que já foram bem

[1] Em inglês: *Architecture Definition Languages (ADLs).*

especificados e testados, diminuindo os custos do processo de codificação do sistema. Entretanto, desenvolver sistemas através da integração de componentes de software existentes pode ser complicado pelos seguintes fatores: introduzir novos ou estender os requisitos de um componente (por exemplo, aumentar a confiabilidade de seus serviços), usar os componentes em um diferente contexto e heterogeneidade de componentes. Utilizando tal abordagem, durante a fase de projeto arquitetural e detalhado, os componentes lógicos do sistema são identificados. Durante a fase de codificação, os componentes de software existentes necessitam ser adaptados para serem compatíveis com os componentes lógicos definidos anteriormente. Desta forma, o desafio para pesquisa futura é a definição de mecanismos eficazes (por exemplo, técnicas de adaptação de componentes [23, 129]) que possibilitem a adição de novas características aos componentes de software existentes.

- Um outro tópico de pesquisa emergente é o desenvolvimento de técnicas de tolerância a falhas que são aptas a lidar com as características de sistemas-de-sistemas [85]. Sistemas-de-sistemas são sistemas complexos construídos a partir de sistemas-componentes existentes que são autônomos (e que possuem objetivos distintos) e que devem prover seus serviços individuais mesmo se o sistema-de-sistemas não está apto no momento a prover os seus. Isto é, um sistema-de-sistemas provê novos serviços, em adição ao serviços providos individualmente por seus sistemas-componentes. Resultados preliminares [11] indicam que a abordagem proposta neste trabalho é válida no contexto do desenvolvimento de sistemas-de-sistemas, contanto que algumas extensões e ajustes sejam feitas nos estilos arquiteturais e padrões de projeto propostos de tal forma que permitam projetistas endereçarem os problemas típicos do desenvolvimento de tais sistemas. Em primeiro lugar, a noção de colaboração torna possível representar explicitamente interações complexas entre existentes sistemas-componentes. Papéis de componentes fornecem uma interface explícita para as novas funcionalidades apresentadas por cada sistema-componente no contexto de um sistema-de-sistemas. Além disso, a abstração de componente tolerante a falhas ideal é recursiva; isto é, cada componente tolerante a falhas ideal pode ser um autônomo sistema-componente. Neste contexto, a interface para o tratamento de exceções deveria ser materializada (isto é, representada explicitamente na arquitetura do sistema) pois torna possível cada sistema-componente implementar seus próprios tratadores de exceções e políticas de recuperação de erros.

- Análises preliminares [11] mostram que o estudo de caso apresentado no Capítulo 7 pode ser estendido através da introdução de sistemas-componentes autônomos que são controlados separadamente mas que são conectados através da malha ferroviária: cidades e aeroportos são conectados pelo sistema ferroviário através de estações. O

desafio para pesquisa futura é desenvolver um sistema que provê suporte às funcionalidades complexas da integração de tais sistemas-componentes. Em nossa opinião, a implementação de um sistema-de-sistemas deveria ser baseada em modernas tecnologias de desenvolvimento baseadas em componentes, tais como CORBA [91] e Enterprise Java Beans [121], que serve como uma plataforma uniforme para tais sistemas heterogêneos.

Apêndice A

Arquivos de Configuração do *Framework*

A arquitetura de software proposta por este trabalho proporciona suporte à codificação de sistemas orientados a objetos confiáveis através da definição de um *framework* orientado a objetos reflexivo. Projetistas utilizam este *framework*: (i) através da definição de novas classes (subclasses) necessárias para uma aplicação específica ou (ii) através da associação de objetos da aplicação a instâncias das meta-classes definidas pelo *framework*.

Esta associação entre objetos e meta-objetos é feita de forma automática desde que os projetistas forneçam alguns arquivos de configuração que são lidos pelo *framework* e que inicializam a meta-configuração associada a uma aplicação específica. O formato destes arquivos de configuração estão descritos neste apêndice.

A.1 Contratos

O primeiro arquivo de configuração possui o formato apresentado na Figura A.1 e descreve a associação entre as classes que implementam as propriedades funcionais de uma aplicação específica e as classes que implementam os detectores de erros (padrão de projeto contrato reflexivo).

Neste arquivo está listado um conjunto de pares ($Componente_i$, $Contrato_i$), onde:

- $Componente_1$ .. $Componente_N$ representam as classes que implementam as propriedades funcionais da aplicação;
- $Contrato_1$ .. $Contrato_N$ representam as classes que implementam os detectores de erros.

Ao ler este arquivo de configuração, o *framework* inicializa a meta-configuração associada à aplicação de tal forma que cada instância de $Componente_i$ será automaticamente

Arquivo de configuração: Contratos

$Componente_1$	$Contrato_1$
$Componente_2$	$Contrato_2$
..	
$Componente_N$	$Contrato_N$

Figura A.1: Contratos

associada ao seu respectivo contrato ($Contrato_i$). Isto é, o meta-nível intercepta a criação de novas instâncias de $Componente_i$ e tratam estas interceptações por associar estas novas instâncias aos seus respectivos contratos ($Contrato_i$).

A.2 Tratamento de exceções

O primeiro arquivo de configuração possui o formato apresentado na Figura A.2 e descreve a associação entre as classes que implementam as propriedades funcionais de uma aplicação específica e as classes que implementam os tratadores de exceções (padrões de projeto estratégia de tratamento de exceções e tratador).

Neste arquivo está listado um conjunto de pares ($Componente_i$, $Excepcional_i$), onde:

- $Componente_1$ .. $Componente_N$ representam as classes que implementam as propriedades funcionais da aplicação;
- $Excepcional_1$ .. $Excepcional_N$ representam as classes que implementam os tratadores para as exceções locais e concorrentes da aplicação.

Arquivo de configuração: Tratamento de exceções

$Componente_1$	$Excepcional_1$
$Componente_2$	$Excepcional_2$
..	
$Componente_N$	$Excepcional_N$

Figura A.2: Tratamento de exceções

Ao ler este arquivo de configuração, o *framework* inicializa a meta-configuração associada à aplicação de tal forma que cada instância de $Componente_i$ será automaticamente

associada ao seu respectivo tratador de exceções ($Excepcional_i$). Isto é, o meta-nível intercepta a criação de novas instâncias de $Componente_i$ e tratam estas interceptações por associar estas novas instâncias aos seus respectivos tratadores de exceções ($Excepcional_i$).

A.3 Redundância de software

O terceiro arquivo de configuração possui o formato apresentado na Figura A.3 e descreve a associação entre uma classe que define serviços tolerante a falhas e as classes que implementam as diferentes versões para os serviços providos pela primeira classe (padrão redundância de software).

Neste arquivo está listado um conjunto dos seguintes elementos:

- $Componente_1$ .. $Componente_N$ representam as classes que definem os serviços tolerantes a falhas;
- BR ou NV identificam a técnica de tolerância a falhas utilizada (Bloco de Recuperação ou N-Versões).
- $Versao_{11}$ .. $Versao_{NZ}$ implementam as diferentes versões dos serviços providos por $Componente_1$ .. $Componente_N$.
- V_i representa o número de versões dos serviços providos por $Componente_i$.

Arquivo de configuração: Redundância de software

$Componente_1\ RB\ V_1$
$Versao_{11}\ Versao_{12}$ .. $Versao_{1V_1}$
$Componente_2\ NV\ V_2$
$Versao_{21}\ Versao_{22}$ .. $Versao_{2V_2}$
..
$Componente_N\ RB\ V_N$
$Versao_{N1}\ Versao_{N2}$ .. $Versao_{NV_N}$

Figura A.3: Redundância de software

Ao ler este arquivo de configuração, o *framework* inicializa a meta-configuração associada à aplicação de tal forma que cada instância de $Componente_i$ será automaticamente associada aos componentes redundantes que implementam as diferentes versões para os serviços providos por $Componente_i$.

A.4 Papéis

O quarto arquivo de configuração possui o formato apresentado na Figura A.4 e descreve a associação entre componentes e seus respectivos papéis de componentes (padrão papel reflexivo).

Neste arquivo está listado um conjunto dos seguintes elementos:

- $Componente_1$ .. $Componente_N$ representam os componentes que desempenham diferentes papéis de componentes em diferentes contextos.
- $Papel_{11}$ .. $Papel_{NZ}$ representam os papéis de componentes.
- P_i representa o número de papéis de componentes desempenhados por $Componente_i$.

Arquivo de configuração: Papéis

$Componente_1\ P_1$
$Papel_{11}\ Papel_{12}$ .. $Papel_{1P_1}$
$Componente_2\ P_2$
$Papel_{21}\ Papel_{22}$ .. $Papel_{2P_2}$
..
$Componente_N\ P_N$
$Papel_{N1}\ Papel_{N2}$ .. $Papel_{NP_N}$

Figura A.4: Papéis

Ao ler este arquivo de configuração, o *framework* inicializa a meta-configuração associada à aplicação de tal forma que cada instância de $Componente_i$ será automaticamente associada aos "objetos papéis", cada um representando um papel de componente que este componente desempenha nos diferentes contextos.

Vale a pena salientar que se $Componente_i$ está listado em mais de um arquivo de configuração, o *framework* associará as novas instâncias deste componente ao meta-objeto chamado delegador que delega operações operações e resultados para outros meta-objetos (Ver discussão seção 5.1.5).

A.5 Controle de estações ferroviárias

Nós utilizamos o *framework* orientado a objetos na codificação deste estudo de caso. Figura A.5 apresenta os arquivos de configuração utilizados na inicialização da meta-configuração associada a este estudo de caso:

- O primeiro arquivo de configuração foi utilizado para descrever a as classes que implementam as atividades de detecção de erros do componente Trem e de seus respectivos papéis de componentes (Figura 7.10).
- O segundo arquivo de configuração foi utilizado para descrever a as classes que implementam as atividades de tratamento de exceções do componente Trem e de seus respectivos papéis de componentes.
- O terceiro arquivo de configuração foi utilizado para descrever a associação entre o componente Trem e seus respectivos papéis de componentes.

Arquivo de configuração: Contratos

Trem TremContrato
Transportador TransportadorContrato
Conexao ConexaoContrato

Arquivo de configuração: Tratamento de exceções

Trem TremExcepcional
Transportador TransportadorExcepcional
Conexao ConexaoExcepcional

Arquivo de configuração: Papéis

Trem 2
Transportador Conexao

Figura A.5: Arquivos de configuração: Controle de estações ferroviárias

A.6 Aplicação bancária

Nós utilizamos o *framework* orientado a objetos na codificação deste estudo de caso. Figura A.6 apresenta os arquivos de configuração utilizados na inicialização da meta-configuração associada a este estudo de caso:

- O primeiro arquivo de configuração foi utilizado para descrever a a classe que implementa as atividades de detecção de erros do componente Conta (Figura 8.2).
- O segundo arquivo de configuração foi utilizado para descrever as classes que implementam as diferentes versões para os serviços providos pelos papéis de componentes do componente Pessoa (Figura 8.6).
- O terceiro arquivo de configuração foi utilizado para descrever a associação entre o componente Pessoa e seus respectivos papéis de componentes (Figura 8.3).

Arquivo de configuração: Contratos

Conta ContaContrato

Arquivo de configuração: Redundância de software

Parceiro RB 4
Zero Vinte Quarenta Restante
Sacador RB 4
Cem Oitenta Sessenta Todo

Arquivo de configuração: Papéis

Pessoa 2
Parceiro Sacador

Figura A.6: Arquivos de configuração: Aplicação bancária

A.7 Protocolo otimista de assinatura de contratos

Nós utilizamos o *framework* orientado a objetos na codificação deste estudo de caso. Figura A.7 apresenta os arquivos de configuração utilizados na inicialização da meta-configuração associada a este estudo de caso.

- O primeiro arquivo de configuração foi utilizado para descrever as classes que implementam as atividades de tratamento de exceções do componente Pessoa e de seus respectivos papéis de componentes (Figura 8.12).
- O segundo arquivo de configuração foi utilizado para descrever a associação entre o componente Pessoa e seus respectivos papéis de componentes (Figura 8.10).

Arquivo de configuração: Tratamento de exceções

Pessoa PessoaExcepcional
Originador OriginadorExcepcional
Receptor ReceptorExcepcional
Terceiro TerceiroExcepcional

Arquivo de configuração: Papéis

Pessoa 3
Originador Receptor Terceiro

Figura A.7: Arquivos de configuração: Protocolo de assinatura de contratos

Bibliografia

[1] A. ALBANO, R. BERGAMINI, G. GHELLI, & R. ORSINI. An Object Data Model with Roles. Em *18th International Conference on Very Large Data Bases*, páginas 39–51, Dublin, Irlanda, Agosto 1993.

[2] R. ALLEN & D. GARLAN. Formalizing Architectural Connection. Em *16th IEEE International Conference on Software Engineering (ICSE'94)*, páginas 71–80, Itália, Maio 1994.

[3] N. ASOKAN. *Fairness in Electronic Commerce.* Tese de Doutorado, University of Waterloo, Ontario, Canadá, 1998.

[4] A. AVIZIENIS. The N-version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, Dezembro 1985.

[5] R. BALTER, S. LACOURTE, & M. RIVEILL. The Guide Language. *The Computer Journal*, 7(6):519–530, 1994.

[6] J.G.P. BARNES. *Programming in Ada.* Addison-Wesley, 1984.

[7] D. BÄUMER, D. RIEHLE, W. SIBERSKI, & M. WULF. Role Pattern. Em *Pattern Languages of Program Design 4*, capítulo 2, páginas 15–32. Addison-Wesley, 2000.

[8] D.M. BEDER, L.E. BUZATO, & C.M.F. RUBIRA. Mecanismos de Tratamento de Exceções para a Construção de Software Tolerante a Falhas. Em *I Simpósio Regional de Tolerância a Falhas*, páginas 99–106, Porto Alegre, Rio Grande do Sul, Brasil, Dezembro 1996.

[9] D.M. BEDER, L.E. BUZATO, & C.M.F. RUBIRA. Exceções e a Construção de Programas Confiáveis. Em *II Simpósio Brasileiro de Linguagens de Programação (SBLP'97)*, páginas 231–244, Campinas, Brasil, Setembro 1997.

[10] D.M. BEDER, L.L. FERREIRA, & C.M.F RUBIRA. Uma Abordagem Reflexiva baseada no modelo de Componente Tolerante a Falhas Ideal para o Desenvolvimento

de Sistemas Orientados a Objetos Confiáveis. Relatório Técnico IC-99-17, Instituto de Computação, Universidade Estadual de Campinas, Brasil, Julho 1999.

[11] D.M. BEDER, B. RANDELL, A. ROMANOVSKY, & C.M.F. RUBIRA. On Applying Coordinated Atomic Actions and Dependable Software Architectures in Developing Complex Systems. Em *4th IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC'01)*, páginas 103–112, Magdeburg, Alemanha, Maio 2001.

[12] D.M BEDER, A. ROMANOVSKY, B. RANDELL, C.R. SNOW, & R.J. STROUD. An Application of Fault Tolerance Patterns and Coordinated Atomic Actions to a Problem in Railway Scheduling. *ACM Operating System Review*, 34(4):21–31, Outubro 2000.

[13] D.M. BEDER & C.M.F. RUBIRA. Comparative Study of Fault-Tolerant Concurrent Mechanisms: Atomic Actions, Conversations and Coordinated Atomic Actions. Relatório Técnico IC-99-21, Instituto de Computação, Universidade Estadual de Campinas, Brasil, Setembro 1999.

[14] D.M. BEDER & C.M.F. RUBIRA. Uma Abordagem Reflexiva baseada em Padrões de Projeto para o Desenvolvimento de Aplicações Distribuídas Confiáveis. Em *VIII Simpósio de Computação Tolerante a Falhas (SCTF'99)*, páginas 152–166, Campinas, Brasil, Julho 1999.

[15] D.M. BEDER & C.M.F. RUBIRA. A Meta-Level Software Architecture based on Patterns for Developing Dependable Collaboration-based Designs. Em *II Workshop Brasileiro de Tolerância a Falhas (WTF'00)*, páginas 34–39, Curitiba, Brasil, Julho 2000.

[16] M. BENVENISTE & V. ISSARNY. Concurrent Programming Notations in the Object-Oriented Language Arche. Relatório Técnico 1882, IRISA/INRIA-Rennes, França, Dezembro 1992.

[17] P.A. BERNSTEIN, V. HADZILACOS, & N. GOODMAN. *Concurrency Control and Recovery in Database Systems.* Addisson-Wesley Publishing Company, 1987.

[18] B.W. BOEHM. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[19] G. BOOCH. *Object-Oriented Design with Applications.* Benjamin/Cummings, 1991.

[20] G. BOOCH, J. RUMBAUGH, & I. JACOBSON. *The Unified Modeling Language User Guide.* Addison Wesley, 1999.

[21] A. BORGIDA. Language Features for Flexible Handling of Exceptions in Information Systems. *ACM Transactions on Database Systems*, 10(4):565–603, 1985.

[22] A. BORGIDA. Exceptions in Object-Oriented Languages. *ACM Sigplan Notices*, 21(10):107–119, 1986.

[23] J. BOSCH. Adapting Object-Oriented Components. Em *ECOOP'97 Workshop on Component-Oriented Programming (WCOP'97)*, páginas 13–21, Jyväskylä, Finlândia, Junho 1997.

[24] F. BUSCHMANN, R. MEUNIER, H. ROHNERT, P. SOMMERLAD, & M. STAL. *A System of Patterns*. John Wiley & Sons, 1996.

[25] L.E. BUZATO. Tolerância a Falhas, Orientação a Objetos e Ações Atômicas. Em *VI Simpósio de Computadores Tolerantes a Falhas*, páginas 33–47, Canela, Rio Grande do Sul, Brasil, Julho 1995.

[26] L.E. BUZATO, C.M.F. RUBIRA, & M.L.B. LISBÔA. A Reflective Object-Oriented Architecture for Developing Fault-Tolerant Software. *Journal of the Brazilian Computer Society*, 4(2):39–48, Novembro 1997.

[27] R.H. CAMPBELL & B. RANDELL. Error Recovery in Asynchronous Systems. *IEEE Transactions on Software Engineering*, SE-12(8):811–826, Agosto 1986.

[28] M. CARILLO-CASTELLON, J. GARCIA-MOLINA, E. PIMENTEL, & I. REPISO. Design by Contract in Smalltalk. *Journal of Object-Oriented Programming*, páginas 23–28, Novembro 1996.

[29] T.H. CORMEN, C.E. LEISERSON, & R.L. RIVEST. *Introduction to Algorithms*. McGraw-Hill, 1990.

[30] F. CRISTIAN. Exception Handling and Software Fault Tolerance. *IEEE Transactions on Computers*, C-31(6):531–540, Junho 1982.

[31] F. CRISTIAN. *Dependability of Resilient Computers*, capítulo Exception Handling, páginas 68–97. Blackwell Scientific Publications, 1989.

[32] Q. CUI & J. GANNON. Data-Oriented Exception Handling. *IEEE Transactions on Software Engineering*, 18(5):393–401, Maio 1992.

[33] F. DANIELS, K. KIM, & M. VOUK. The Reliable Hybrid Pattern - a Generalized Software Fault Tolerant Design Pattern. Em *4th Pattern Languages of Programming Conference (PloP'97)*, 1997.

[34] C.T. DAVIES. Data Processing Spheres of Control. *IBM System Journal*, 17(2):179–198, 1978.

[35] M. DE CHAMPLAIN. The Contract Pattern. Em *4th Pattern Languages of Programming Conference (PloP'97)*, 1997.

[36] R. DE LEMOS. A Co-operative Object-Oriented Architecture for Adaptive Systems. Em *7th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, páginas 120–128, Escócia, Abril 2000.

[37] E.W. DIJKSTRA. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[38] C. DONY. An Object-Oriented Exception Handling System for an Object-Oriented Programming. *Lectures Notes in Computer Science*, 322:146–161, Agosto 1988.

[39] C. DONY. Exception Handling and Object-Oriented Programming: Towards a Synthesis. *ACM Sigplan Notices*, 25(10):322–330, 1990.

[40] D.F. D'SOUZA & A.C. WILLS. *Object, Components and Frameworks with UML: The Catalysis Approach.* Addisson-Wesley, 1998.

[41] A. ELIENS. *Principles of Object-Oriented Software Development.* Addison-Wesley, 1995.

[42] J.L. EPPINGER, L.M. MUMMERT, & A.Z. SPECTOR, editores. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann Publishers, Inc., 1991.

[43] J.-C. FABRE & T. PÉRENNOU. A MetaObject Architecture for Fault-Tolerant Distributed Systems: the FRIENDS Approach. *IEEE Transactions on Computers*, 47(1):78–95, 1998.

[44] J. FERBER. Computational Reflection in Class-based Object-Oriented Languages. Em *4th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, páginas 317–326, New Orleans, Louisiana, EUA, 1989.

[45] L.L. FERREIRA & C.M.F. RUBIRA. Integration of Fault Tolerance Techniques: a System of Pattern to Cope with Hardware, Software and Environmental Fault Tolerance. Em *Digest of FastAbstracts: FTCS'98*, páginas 25–26, Munich, Junho 1998.

[46] L.L. FERREIRA & C.M.F. RUBIRA. Padrão State Reflexivo: Refinamento do Padrão de Projeto State para uma Arquitetura Reflexiva. Em *XII Simpósio Brasileiro de Engenharia de Software - (SBES'98)*, Outubro 1998.

[47] D.G. FIRESMITH. Frameworks: the Golden Path to Object Nirvana. *Journal of Object-Oriented Programming*, 6(6):6–8, Novembro 1993.

[48] D. FLANAGAN. *Java in a Nutshell.* O'Reilly & Associates, 3 edição, 1999.

[49] E. GAMMA, R. HELM, R. JOHNSON, & J. VLISSIDES. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison Wesley Publishing Company, 1995.

[50] B. GARBINATO, R. GUERRAOUI, & K. MAZOUNI. Implementation of the GARF Replicated Objects Plataform. *Distributed Systems Engineering Journal*, 2:14–27, 1995.

[51] A. GARCIA. Tratamento de Exceções em Sistemas Concorrentes Orientado a Objetos. Dissertação de Mestrado, Instituto de Computação - Universidade Estadual de Campinas, Brasil, Março 2000.

[52] A.F. GARCIA, D.M. BEDER, & C.M.F. RUBIRA. An Object-Oriented Exception Handling Mechanism for Developing Dependable Component-Based Object-Oriented Software. Em *X International Symposium on Software Reliability Engineering*, Florida, USA, Novembro 1999.

[53] A.F. GARCIA, D.M. BEDER, & C.M.F. RUBIRA. An Exception Handling Software Architecture for Developing Fault-Tolerant Software. Em *V International Simposium on High Assurance Systems Engineering (HASE'00)*, Albuquerque, New Mexico, EUA, 2000.

[54] A.F. GARCIA, D.M. BEDER, & C.M.F. RUBIRA. Unified Meta-Level Software Architecture for Sequential and Concurrent Exception Handling. *Computer Journal, Special Issue on High Assurance Systems Engineering*, 2001.

[55] D. GARLAN, R. ALLEN, & J. OCKERBLOOM. Exploiting Style in Architectural Design Environments. Em *SIGSOFT'94: Foundations of Software Engineering*, páginas 175–188, New Orleans, Louisiana, EUA, Dezembro 1994.

[56] C. GHEZZI, M. JAZAYERI, & D. MANDRIOLI. *Fundamentals of Software Engineering.* Prentice-Hall, 1991.

[57] G. GOTTLOB, M. SCHREFL, & B. RÖCK. Extending Object-Oriented Systems with Roles. *ACM Transactions on Information Systems*, 14(3):268–296, Julho 1996.

[58] J. GRAY & A. REUTER. *Transaction Processing: Concepts and Techniques.* Morgan Kaufman Publishers, San Mateo, California, 1993.

[59] W.L. HEIMERDINGER & C.B. WEISNSTOCK. A Conceptual Framework for System Fault Tolerance. Relatório Técnico CMU/SEI - 92 - TR-33, Carnegie-Mellon University, Pennsylvania, EUA, Outubro 1992.

[60] R. HIRSCHFELD & J. EASTMAN. Lock Server. Em *4th Pattern Languages of Programming Conference (PloP'97)*, 1997.

[61] C.A.R. HOARE. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[62] M. HOF, H. MÖSSENBÖCK, & P. PIRKELBAUER. Zero-Overhead Exception Handling Mechanism. Em *Lectures Notes in Computer Science 1338*, páginas 423–431. Springer-Verlag, 1997.

[63] J.J. HORNING, H.C. LAUER, P.M. MELLIAR-SMITH, & B. RANDELL. A Program Structure for Error Detection and Recovery. Em *Lecture Notes in Computer Science 16*, páginas 171–187. Spring-Verlag, 1974.

[64] V. ISSARNY. An Exception-Handling Mechanism for Parallel-Object-Oriented Programming : Toward Reusable, Robust Distributed Software. *Journal of Object-Oriented Programming*, 6(6):29–40, Outubro 1993.

[65] I. JACOBSON. *Object-Oriented Software Engineering - A Use Case Driven Approach.* Addison-Wesley, 1992.

[66] P. JALOTE. *Fault Tolerance in Distributed Systems.* Prentice-Hall, Inc., 1994.

[67] P. JALOTE & R.H. CAMPBELL. Atomic Actions for Fault-Tolerance Using CSP. *IEEE Transactions on Software Engineering*, SE-12(1):59–68, Janeiro 1986.

[68] K.H. KIM. Approaches to Mechanization of the Conversation Scheme Based on Monitors. *IEEE Transactions on Software Engineering*, SE-8(3):189–197, Maio 1982.

[69] R. KOO & S. TOUEG. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, Janeiro 1987.

[70] S. KRAKOWIAK, M. MEYSEMBOURG, H. NGUYEN, M. RIVELLI, C. ROISIN, & X. ROUSSET DE PINA. Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications. *Journal of Object-Oriented Programming*, 3(3), Setembro 1990.

[71] B.B. KRISTENSEN. Object-Oriented Modelling with Roles. Em *2nd International Conference on Object Oriented Information Systems (OOIS'95)*, 1995.

[72] P. LALANDA & S. CHERKI. Object-Orientation and Software Architecture. Em *ECOOP'98 Workshop on Object-Oriented Software Architectures*, páginas 115–119, Belgium, Julho 1998.

[73] J.C. LAPRIE. Dependable Computing and Fault Tolerance: Concepts and Terminology. Em *15th International Symposium on Fault Tolerant Computing Systems (FTCS'15)*, páginas 2–11, Junho 1985.

[74] P.A. LEE & T. ANDERSON. *Fault Tolerance: Principles and Practice.* Springer-Verlag, 2 edição, 1990.

[75] T. LEWIS, L. ROSENSTEIN, W. PREE, A. WEINAND, E. GAMMA, P. CALDER, G. ANDERT, J. VLISSIDES, & K. SCHMUCKER. *Object Oriented Application Framework.* Mannings Publication Co., 1995.

[76] M.L.B. LISBÔA. A New Trend on the Development of Fault-Tolerant Applications: Software Meta-Level Architectures. Em *IFIP International on Dependable Computing and its Applications*, Janeiro 1998.

[77] M.L.B. LISBÔA & P.A. AZEREDO. Paradigmas de Linguagens de Programação: o enfoque de Tolerância a Falhas. Em *XIII Jornada de Atualização em Informática*, Caxambú, Minas Gerais, Brasil, 1994.

[78] B. LISKOV. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, Março 1988.

[79] B.H. LISKOV & A. SNYDER. Exception Handling in CLU. *IEEE Transaction on Software Engineering*, SE-5(6):546–558, Novembro 1979.

[80] D.C. LUCKHAM, J.J. KENNEY, L.M. AUGUSTIN, J. VERA, D. BRYAN, & W. MANN. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, Abril 1995.

[81] M. LYU, editor. *Handbook of Software Reliability Engineering.* McGraw-Hill and IEEE Computer Society Press, New York, 1996.

[82] M. LYU & B. KRISHNAMURTHY, editores. *Software Fault Tolerance.* John Wiley & Son, New York, 1996.

[83] M.D. MACLAREN. Exception Handling in PL/I. *SIGPLAN Notices*, 12(3):101–104, Março 1977.

[84] P. MAES. Concepts and Experiments in Computacional Reflection. *ACM SIGPLAN Notices*, 22(12):147–155, Dezembro 1987.

[85] M. MAIER. Architecting Principles for Systems-of-Systems. *System Engineering*, 1(4):267–284, 1998.

[86] M. MATTSSON. *Evolution and Composition of Object-Oriented Frameworks.* Tese de Doutorado, University of Karlskrona/Ronneby - Department of Software Enginneering and Computer Science, Ronneby, Suécia, 1999.

[87] A.J. MENEZES, P.C. VAN OORSCHOT, & S.A. VANSTONE. *Handbook of Applied Cryptography.* CRC press, 1996.

[88] B. MEYER. *Object-Oriented Software Construction.* New York: Prentice-Hall, 1988.

[89] B. MEYER. *Eiffel: The Language.* Prentice Hall, 1992.

[90] S. MITCHELL, A. BURNS, & A. WELLINGS. MOPping up Exceptions. Em *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, páginas 365–366, 1998.

[91] Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.0*, 1995.

[92] A. OLIVA & L.E. BUZATO. Composition of Meta-Objects in Guaraná. Em *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, páginas 86–90, Vancouver, Canadá, Outubro 1998.

[93] A. OLIVA, I.C. GARCIA, & L.E. BUZATO. The Reflective Architecture of Guaraná. Relatório Técnico IC-98-14, Instituto de Computação, Universidade de Campinas, Abril 1998.

[94] R.S. PRESSMAN. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 5 edição, 2001.

[95] E.M. QUADROS. Uma Abordagem Orientada a Objetos para Programação Distribuída Confiável. Dissertação de Mestrado, Instituto de Computação, Universidade Estadual de Campinas, Junho, 1997.

[96] N.L.V. QUADROS. Um Arcabouço de Software Reflexivo para Persistência de Objetos em Bases de Dados Heterogêneos. Dissertação de Mestrado, Instituto de Computação, Universidade Estadual de Campinas, Julho 1999.

[97] B. RANDELL. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, 1975.

[98] B. RANDELL, A. ROMANOVSKY, R.J. STROUD, J. XU, & A.F. ZORZO. Coordinated Atomic Actions: from Concept to Implementation. Relatório Técnico 595, Department of Computing Science, University of Newcastle upon Tyne, UK, 1997.

[99] B. RANDELL & J. XU. Object-Oriented Software Fault Tolerance: Framework, Reuse and Design Diversity. Em *1st PDCS2 Open Workshop*, páginas 165–184, Toulouse, 1993.

[100] A. ROMANOVSKY. Conversations of Objects. *Computer Languages*, 21(3):147–163, 1995.

[101] A. ROMANOVSKY. Practical Exception Handling and Resolution in Concurrent Programs. *Computer Journal*, 23(7):43–58, 1997.

[102] A. ROMANOVSKY, B. RANDELL, R.J. STROUD, J. XU, & A.F. ZORZO. Implementing Synchronous Coordinated Atomic Actions Based on Forward Error Recovery. Relatório Técnico 561, Department of Computing Science, University of Newcastle upon Tyne, UK, 1997.

[103] A. ROMANOVSKY, J. XU, & B. RANDELL. Exception Handling and Resolution in Distributed Object-Oriented Systems. Em *16th International Conference on Distributed Computing Systems (ICDCS'96)*, páginas 545–553, Hong Kong, 1996.

[104] A. ROMANOVSKY, J. XU, & B. RANDELL. Exception Handling in Object-Oriented Real-Time Distributed Systems. Em *1st International Symposium on Object-oriented Real-time Distributed Computing (ISORC'98)*, Kyoto, Japan, Abril 1998.

[105] C.M.F. RUBIRA. *Structuring Fault-Tolerant Object-Oriented Systems Using Inheritance and Delegation.* Tese de Doutorado, Department of Computing Science, University of Newcastle upon Tyne, Outubro 1994.

[106] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY, & W. LORENSEN. *The Object-Oriented Modeling and Design.* Prentice-Hall, 2 edição, 1992.

[107] M. SHAW, R. DELINE, V. KLEIN, T.L. ROSS, D.M. YOUNG, & G. ZELESNIK. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335, Abril 1995.

[108] M. SHAW & D. GARLAN. *Software Architecture: Perpectives on an Emerging Discipline.* Prentice Hall, 1996.

[109] S.K. SHRIVASTAVA. Concurrent Pascal with Backward Error Recovery. *Software Practice and Experience*, 9:1021–1033, 1979.

[110] S.K. SHRIVASTAVA, G.N. DIXON, & G.D. PARRINGTON. An Overview of the Arjuna Programming System. *IEEE Software*, 8(1):66–73, Janeiro 1991.

[111] S.K. SHRIVASTAVA, L.V. MANCINI, & B. RANDELL. The Duality of Fault-Tolerant System Structures. *Software - Practice and Experience*, 23(7):773–798, Julho 1993.

[112] Y. SMARAGDAKIS & D. BATORY. Implementing Reusable Object-Oriented Components. Em *5th International Conference on Software Reuse (ICSR'98)*, Victoria, Canada, Junho 1998.

[113] B. SMITH. *Reflection and Semantics in a Procedural Language.* Tese de Doutorado, Massachusetts Institute of Technology, 1982.

[114] C. SNOW. Distributed Real-Time Control of a Distributed Model Railway Layout. Relatório Técnico, Department of Computing Science, University of Newcastle (em preparação).

[115] I. SOMMERVILLE. *Software Engineering.* Addison-Wesley, 6 edição, 2001.

[116] A.Z. SPECTOR. Suport for Distributed Transactions in the TABS Prototype. *IEEE Transactions on Software Engineering*, SE-11(6):520–530, Junho 1985.

[117] R.J. STROUD. Transparency and Reflection in Distributed Systems. Em *5th European SIGOPS Workshop on Models and Paradigms for Distributed Systems Structuring*, Mont Saint-Michel, França, Setembro 1992.

[118] R.J. STROUD & Z. WU. Using Meta-objects to Adapt a Persistent Object System to Meet Applications needs. Em *6th SIGOPS European Workshop on Matching Operating Systems to Applications Needs*, 1994.

[119] B. STROUSTRUP. *The C++ Programming Language.* Addison Wesley, 3 edição, 1997.

[120] S. SUBRAMANIAN & W. TSAI. Backup Pattern: Designing Redundancy in Object-Oriented Software. Em *3th Pattern Languages of Programming Conference (PloP'96)*, 1996.

[121] Sun Microsystems. *Enterprise Java Beans*, 1998.

[122] C. SZYPERSKI. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Publishing Company, 1998.

[123] A.S. TANENBAUM. *Computer Networks.* Prentice-Hall, 3 edição, 1996.

[124] P. THOMAS & R. WEEDON. *Object-Oriented Programming in Eiffel.* Addison Wesley, 1995.

[125] E. TRAMONTANA & R. DE LEMOS. A Reflective Approach for Describing Cooperation between Objects. Relatório Técnico 663, University of Newcastle upon Tyne, Inglaterra, Março 1999.

[126] J. VACHON, D. BUCHS, M. BUFFO, G. SERUGENDO, B. RANDELL, A. ROMANOVSKY, R. STROUD, & J. XU. COALA - A Formal Language for Co-ordinated Atomic Actions. Em *3rd Year Report - ESPRIT Long Term Research Project 20072 on Design for Validation*, LAAS, França, Novembro 1998.

[127] J. VACHON, N. GUELFI, & A. ROMANOVSKY. Using COALA to Develop a Distributed Object-Based Application. Em *2nd International Symposium on Distributed Objects and Applications (DAO'00)*, páginas 195–208, 2000.

[128] M. VANHILST & D. NOTKIN. Using Role Components to Implement Collaboration-based Designs. Em *11th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, páginas 359–369, San Jose, California, EUA, Novembro 1996.

[129] G.M. WEISS. Adaptação de Componentes de Software para o Desenvolvimento de Sistemas Confiáveis. Dissertação de Mestrado, Instituto de Computação, Universidade Estadual de Campinas, Junho 2001.

[130] J. XU, B. RANDELL, A. ROMANOVSKY, C.M.F. RUBIRA, R. STROUD, & Z. WU. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. Em *25th International Symposium on Fault Tolerant Computing (FTCS'25)*, páginas 499–509, Pasadena, California, EUA, 1995.

[131] J. Xu, B. Randell, A. Romanovsky, R. Stroud, A. Zorzo, A. Burns, S. Mitchell, & A. Wellings. Cooperative and Competitive Concurrency in Fault-Tolerant Distributed Systems. Em *DeVa 1st Year Report*, páginas 21–42, França, Janeiro 1997.

[132] J. Xu, B. Randell, A. Romanovsky, R. Stroud, A. Zorzo, E. Canver, & F. von Henke. Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions. Em *29th International Symposium on Fault-Tolerant Computing (FTCS'99)*, páginas 68–75, Madison, EUA, 1999.

[133] S. Yemini & D.M. Berry. A Modular Verifiable Exception Handling Mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2):214–243, Abril 1985.

[134] Y. Yokote, F. Teraoka, & M. Tokoro. A Reflective Architectures for an Object-Oriented Distributed System. Em *3rd European Conference on Object-Oriented Programming (ECOOP'89)*, páginas 89–106, Nottingham, Inglaterra, Julho 1989.

[135] A.F. Zorzo. *Multiparty Interactions in Dependable Distributed Systems.* Tese de Doutorado, University of Newcastle upon Tyne, Inglaterra, 1999.

[136] A.F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R.J. Stroud, & I.S. Welch. Using Coordinated Atomic Actions to Design Complex Distributed Object Systems. Em *OOPSLA'97 Workshop on Dependable Distributed Object Systems*, Outubro 1997.

[137] A.F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R.J. Stroud, & I.S. Welch. Using Coordinated Atomic Actions to Design Complex Safety-Critical Systems: The Production Cell Case Study. *Software: Practice & Experience*, 29(7):1–21, 1997.