O conteúdo do presente relatório é de única responsabilidade do(s) autor(es).
The contents of this report are the sole responsibility of the author(s).

Uma abordagem reflexiva baseada no modelo de componente tolerante a falhas ideal para o desenvolvimento de sistemas orientados a objetos confiáveis

Delano M. Beder *Luciane L. Ferreira*
Cecília M.F. Rubira

Relatório Técnico IC–99-17

Julho de 1999

Uma abordagem reflexiva baseada no modelo de componente tolerante a falhas ideal para o desenvolvimento de sistemas orientados a objetos confiáveis

Delano M. Beder Luciane L. Ferreira Cecília M.F. Rubira

e-mail: {delano,972311,cmrubira}@dcc.unicamp.br

Resumo

Tolerância a falhas é um importante requisito no desenvolvimento de sistemas computacionais modernos. Entretanto, a contrução de sistemas tolerantes a falhas é uma atividade complexa especializada durante todo o ciclo de desenvolvimento do sistema. Além disso, idealmente a provisão de tolerância a falhas deveria ser feita de forma estruturada e não-intrusiva. Neste contexto, propomos um conjunto de padrões (decisões de projetos) que auxiliam o desenvolvimento de sistemas orientados a objetos tolerantes a falhas. Estes padrões são baseados no modelo de componente tolerante a falhas ideal. O conceito de reflexão computacional é utilizado objetivando a obtenção de uma separação explícita entre o comportamento normal e anormal dos componentes. As atividades normais, que implementam os requisitos funcionais da aplicação, são implementados no nível base, enquanto as atividade anormais, que implementam os mecanismos de tolerância a falhas, são implementados no meta-nível, através de meta-objetos. Os padrões representam decisões de projeto que devem ser tomadas no desenvolvimento de sistemas orientados a objetos tolerantes a falhas objetivando a obtenção de um projeto estruturado, onde a introdução dos mecanismos de tolerância a falhas é feita de forma controlada e consistente.

Abstract

Fault tolerance represents a major challenge to design of moderns computing systems. However, the construction of fault-tolerant systems is not a simple task. It requires the use of appropriate techniques during the whole software development cycle. Besides, ideally fault-tolerance mechanisms should be incorporated to the original system in a structured and non-intrusive manner. In this context, we propose a set of patterns (design decisions) that make easier the task of building fault-tolerant object-oriented systems. This patterns are based on idealised fault-tolerant component model. The computacional reflection concept is used aiming a clear separation of the normal activity from the abnormal activity of a software component. The normal activities, that implement the functional requirements, are implemented by base-level objects, while the abnormal activities, that implement the fault tolerance mechanisms, are implemented by meta-level objects. The patterns are design choices that should be taken during the software development cycle aiming a structured design, where the fault-tolerance mechanisms is incorporated to the original system in a structured and controlled manner.

1 Introdução

Muitas aplicações modernas requerem alto grau de tolerância a falhas, disponibilidade, como exemplo, centrais telefônicas, bancos de dados, sistemas distribuídos, etc. E os usuários dessas aplicações esperam que falhas sejam toleradas sem deteriorar a funcionalidade e confiabilidade do sistema. Entretanto, a contrução de sistemas tolerantes a falhas é uma tarefa complexa especializada durante todo o ciclo de desenvolvimento do sistema. Além disso, idealmente a provisão de tolerância a falhas deveria ser feita de forma estruturada e não-intrusiva.

Neste contexto, propomos um conjunto de padrões (decisões de projetos) que auxiliam o desenvolvimento de sistemas orientados a objetos tolerantes a falhas. Estes padrões são baseados no modelo de componente tolerante a falhas ideal definido por Lee e Anderson [6]. Um componente tolerante a falhas ideal possui uma separação clara entre o seu comportamento normal e anormal, e uma interface bem definida que inclui as respostas normais e excepcionais que podem ser retornadas pelos seus serviços.

O conjunto de padrões também utiliza o conceito de contrato [4, 7], não apenas na especificação de um componente tolerante a falhas ideal mas também no projeto e implementação do mesmo. O objetivo é garantir que os requisitos de tolerância a falhas nos componentes sejam consistentes em todas as fases do ciclo de desenvolvimento do software orientado a objetos. O conceito de reflexão computacional é utilizado objetivando a obtenção de uma separação explícita entre o comportamento normal e anormal do componente. As atividades normais, que implementam os requisitos funcionais da aplicação, são implementados no nível base, enquanto as atividade anormais, que implementam os mecanismos de tolerância a falhas, são implementados no meta-nível, através de meta-objetos. A visão a ser adotada neste trabalho é que deveria ser possível projetar um sistema concentrando-se apenas nos seus requisitos funcionais e, então, integrar os componentes responsáveis pelos requisitos não-funcionais do sistema sem prejudicar a arquitetura do sistema original. A obtenção dessa separação nítida entre os requisitos funcionais e não-funcionais do sistema pode reduzir a complexidade do sistema final, e também facilitar sua extensão e manutenção. A técnica de reflexão computacional pode ser de grande importância para a obtenção dessa transparência desejada pois permite alterar o modo como um programa é executado, sem alterar o que o programa faz.

Os padrões representam decisões de projeto que devem ser tomadas no desenvolvimento de sistemas orientados a objetos tolerantes a falhas objetivando a obtenção de um projeto estruturado, onde a introdução dos mecanismos de tolerância a falhas é feita de forma controlada e consistente, em todas as fases do ciclo de desenvolvimento do software. Os padrões são fortemente relacionados, e cada um resolve um problema dentro de um contexto particular em cada fase de desenvolvimento dos componentes da aplicação.

Na seção 2 apresentamos o conceito de componente tolerante a falhas ideal definido por Lee e Anderson [6]. Na seção 3 apresentamos algumas técnicas orientadas a objetos para a estruturação de software tais como padrões de projeto e reflexão computacional [3]. A utilização de tais técnicas possibilita um alto grau de reusabilidade, melhor estruturação e menor custo no desenvolvimento das aplicações tolerantes a falhas. Na seção 4 apresentamos o conjunto de decisões de projeto proposto que auxilia o desenvolvimento de sistemas

orientadas a objetos tolerantes a falhas. Na seção 5 apresentamos um exemplo do uso dos padrões na implementação de um componente tolerante a falhas ideal. E finalmente, na seção 6 sintetizamos as principais idéias deste trabalho e apontamos nossos trabalhos futuros.

2 Componente tolerante a falhas ideal

Seguindo a terminologia definida por Lee e Anderson [6], um sistema consiste de um conjunto de componentes que interagem sobre o controle de um projeto. Componentes por sua vez também podem ter sub-componentes. O projeto do sistema também é um componente, mas com características especiais, como exemplo, a responsabilidade de controlar a interação entre os componentes do sistema. Componentes recebem requisições de serviços e produzem respostas a estas requisições (figura 1). Com o intuito de produzir respostas a uma dada requisição de serviço, um componente pode solicitar a seus sub-componentes um determinado serviço.

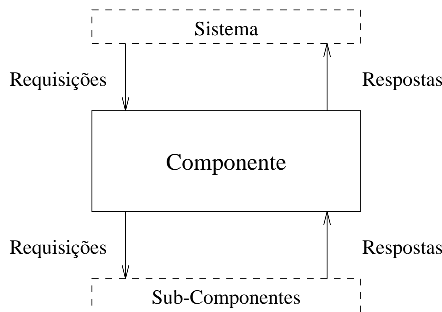

Figura 1: Componente geral de um sistema

As respostas de um componente podem ser particionadas em duas categorias: normais e anormais. Respostas normais são aquelas que um componente produz quando tudo ocorre corretamente e as respostas anormais são aquelas produzidas quando algo anormal ocorreu durante a execução do componente. As respostas anormais de um componente são denominadas de exceções e significa 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 particionado em atividade normal e atividade anormal (tratamento de exceções). A atividade normal implementa o serviço normal do componente enquanto a atividade anormal implementa as medidas para tolerar as falhas que causaram as exceções. Esta separação entre a atividade normal e anormal leva-nos à definição do componente tolerante a falhas ideal [6] (figura 2).

Exceções podem ser classificadas em três grupos: exceções de interface, exceções internas e exceções de falha. Exceções de interface são sinalizadas em resposta a uma requisição de um serviço não disponível na interface do componente. Uma possível causa é uma falha no projeto do componente. Exceções internas são geradas quando um componente detecta

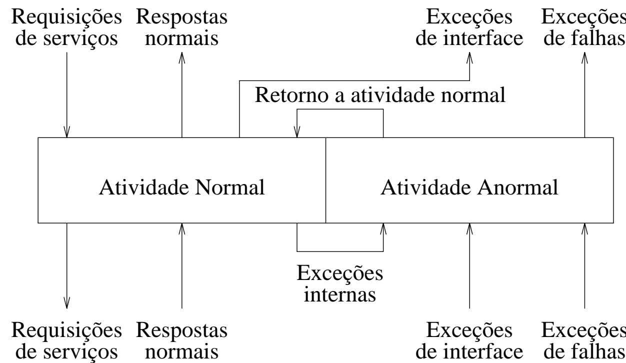

Figura 2: Descrição de um componente tolerante a falhas ideal

uma situação inesperada durante sua atividade normal. Se esta exceção não pode ser tratada pelo próprio componente, então uma exceção de falha 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 falha) de uma invocação de um outro componente ou detecta uma condição anormal (exceção interna) durante sua execução normal, ele invoca os apropriados mecanismos 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 falha é sinalizada.

3 Técnicas OO para estruturação de software

O modelo de objetos apresenta-se como um modelo promissor para o desenvolvimento de software tolerante a falhas devido as características inerentes ao próprio modelo de objetos, tais como, abstração de dados, encapsulamento, herança e reutilização de software. O uso de técnicas orientadas a objetos facilita o controle da complexidade do sistema porque promove uma melhor estruturação de seus componentes. Além disso, há uma forte correspondência entre o modelo de componentes descrito na seção 2 e o modelo de objetos. Da mesma forma que componentes, objetos têm uma interface externa bem definida que fornece operações que manipulam e encapsulam estados internos. Devido a estas semelhanças, componentes podem ser tratados como objetos [6]. Entretanto, o modelo de objetos difere do modelo de componentes pelo uso de conceitos como herança e polimorfismo.

A seguir, apresentamos algumas técnicas orientadas a objetos para estruturação de software utilizadas na construção de componentes tolerantes a falhas. Estas técnicas são utilizadas objetivando: um alto grau de reusabilidade, melhor estruturação e menor custo no desenvolvimento das aplicações distribuídas.

3.1 Padrões de projeto e reflexão computacional

Padrões permitem que soluções de software previamente testadas sejam reutilizadas, permitindo o desenvolvimento de aplicações flexíveis, elegantes e reutilizáveis. Eles capturam a experiência dos projetistas, permitindo que a mesma seja repassada a outros, aumentando

o grau de reutilização de uma aplicação. Um padrão provê soluções em diferentes graus de abstração, desde linguagens de programação até arquitetura de sistemas e têm sido utilizados nos mais variados contextos. Um exemplo é o padrão reflexão [3] que captura o conceito de reflexão computacional.

Neste padrão, uma aplicação é dividida em duas partes: uma que provê informação sobre propriedades selecionadas do próprio sistema (o meta-nível) e outra que implementa a lógica da aplicação (o nível base). Em geral, no nível base de uma aplicação, existem componentes que colaboram entre si para oferecerem ao usuário as funcionalidades do sistema. O meta-nível consiste de um conjunto meta-objetos. Cada meta-objeto encapsula informação selecionada acerca de um único aspecto da estrutura, comportamento, ou estado do nível base.

Um protocolo de meta-objetos estabelece uma interface entre os componentes do nível base e os componentes do meta-nível. A transparente associação entre o meta-nível e o nível base é feita através de mecanismos de interceptação. Cada pedido de serviço enviado para um componente, é interceptado e dirigido ao seu meta-objeto e este fica encarregado da execução do serviço. O objeto emissor (cliente) do pedido de serviço não toma conhecimento da computação reflexiva: ele envia pedidos de serviços para componentes e recebe o resultado esperado, sem saber que o pedido de serviço foi desviado para um meta-objeto. A figura 3 ilustra tal comportamento.

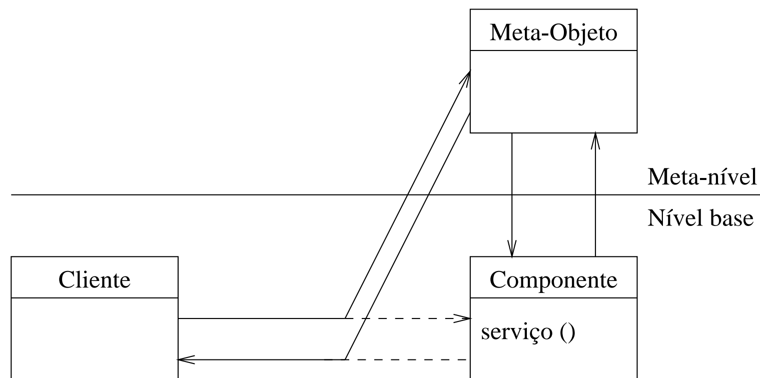

Figura 3: Reflexão computacional

A visão a ser adotada neste trabalho é de que se deveria ser possível projetar um sistema concentrando-se apenas nos seus requisitos funcionais e, então, integrar os componentes responsáveis pelos requisitos não-funcionais do sistema sem prejudicar a arquitetura do sistema original. Um requisito não-funcional é uma característica de um sistema que não está definida na sua descrição funcional. Geralmente, estes requisitos endereçam aspectos relacionados à confiabilidade, eficiência, segurança entre outros. A obtenção dessa separação nítida entre os requisitos funcionais do sistema e seus requisitos não-funcionais pode reduzir a complexidade do sistema final, e também facilitar sua extensão e manutenção. A técnica de reflexão computacional pode ser de grande importância para a obtenção dessa transparência desejada pois permite alterar o modo como um programa é interpretado, sem alterar o que o programa faz.

4 Decisões de projeto

Como mencionado anteriormente, os padrões correspondem a decisões de projeto que devem ser tomadas durante todas as fases de desenvolvimento de um sistema crítico que utiliza o modelo de componente tolerante a falhas ideal. A descrição de cada padrão inclui:

- A decisão de projeto (ou padrão), que resume o que deve ser feito dentro de um contexto particular de desenvolvimento.
- O problema que ocorre em um determinado contexto, incluindo as restrições e exigências que devem ser atendidas pela solução.
- A solução detalhada do problema.
- As conseqüências, positivas e negativas, obtidas com aplicação do padrão.

4.1 Utilize contratos na especificação dos componentes

Problema: 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 pré-condições 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 de cada instância que devem ser respeitadas na execução de qualquer serviço. Qualquer violação destas condições deve gerar uma resposta excepcional na execução de um serviço.

Solução: A especificação de um componente tolerante a falhas ideal pode ser feita utilizando a metodologia de projeto por contrato introduzida por Meyer [7]. Um contrato define os aspectos observáveis pelos clientes de um componente de software. Contratos são definidos através de pré-condições e pós-condições e invariantes de classes. Se uma destas condições é violada, então uma exceção é levantada. Um contrato especifica obrigações e direitos dos **contratuentes**: o cliente e o servidor. Ele especifica quais são as restrições sobre os parâmetros de entrada de um serviço que devem ser satisfeitas pelo cliente, ou seja, as pré-condições, 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 disto, especifica as invariantes da classe, que são todas as restrições impostas sobre o estado interno de todas as instâncias da classe.

Conseqüências: A utilização de contratos na especificação de um componente tolerante a falhas ideal obriga à determinação rigorosa das condições excepcionais do componente nas primeiras fases de desenvolvimento do software. Todas as fases subseqüentes devem seguir

esta especificação, obtendo-se assim mais consistência entre as fases de desenvolvimento, diminuindo-se também erros devidos à omissão dos possíveis casos excepcionais e dos seus respectivos tratamentos.

4.2 Defina uma classe Contrato que faz a transição da especificação para o projeto dos componentes

Problemas: A metodologia de projeto por contrato tem como suporte a linguagem Eiffel [7], que implementa a verificação de pré e pós-condições e as invariantes da classe embutidos na própria linguagem. O problema está em fazer o mapeamento dos contratos definidos na especificação para o projeto e implementação de classes, considerando-se linguagens que não fornecem este tipo de suporte. Idealmente, a implementação do contrato deve ser feita de uma forma não intrusiva da classe do componente, para não complicar a implementação dos métodos da mesma, evitando-se uma programação defensiva (com um número excessivo de testes). Isto facilita o entendimento e manutenção da classe do componente, que implementa os seus serviços sem se preocupar com o cumprimento do contrato.

Um outro problema que deve ser considerado no projeto e implementação de contratos é o problema de herança restrita (subtipo), onde uma subclasse deve **honrar** o contrato da sua superclasse. O subcontrato de uma classe derivada deve atender às pré e pós-condições e as invariantes da classe estabelecidos no contrato da superclasse, podendo tornar as pré-condições mais **leves**, as pós-condições mais **restritas**, e a invariante da classe deve ser **mantida**, podendo-se acrescentar condições para as instâncias da subclasse. Quando a linguagem de programação não dá suporte para a implementação de contratos e subcontratos, o projeto e a implementação destas restrições deve ser feito de uma forma sistemática e a mais explícita possível.

Solução: Para o mapeamento de um contrato definido na especificação para o projeto e implementação das classes dos componentes ideais, defina uma classe que representa o contrato. A classe **Contrato** define métodos que implementam os testes de invariantes da classe, pré e pós-condições para cada serviço crítico do componente, retornando exceções caso as condições não sejam verificadas. Na classe **Contrato**, para cada serviço definido na classe do componente, existe um método correspondente que verifica as pré-condições, testando os parâmetros de entrada, podendo também testar condições relacionadas com o estado interno do objeto antes de um serviço ser executado (figura 4). Do mesmo modo, para cada serviço do componente, existe um método que testa as pós-condições que devem ser satisfeitas pelo resultado do serviço, podendo testar também estados internos do objeto depois que um serviço é executado. Além disto, existe um método que testa as invariantes da classe, que correspondem às condições sobre o estado interno de todas as instâncias da classe. Todos os métodos da classe **Contrato** devem receber como parâmetro uma referência para o objeto do componente ideal, para que possam fazer os testes sobre o estado interno do objeto.

Todos os métodos da classe **Contrato** retornam como resultado um valor booleano verdadeiro, caso as condições sejam satisfeitas, ou retornam exceções, caso contrário. Os métodos que testam pré-condições retornam exceções que correspondem às exceções de in-

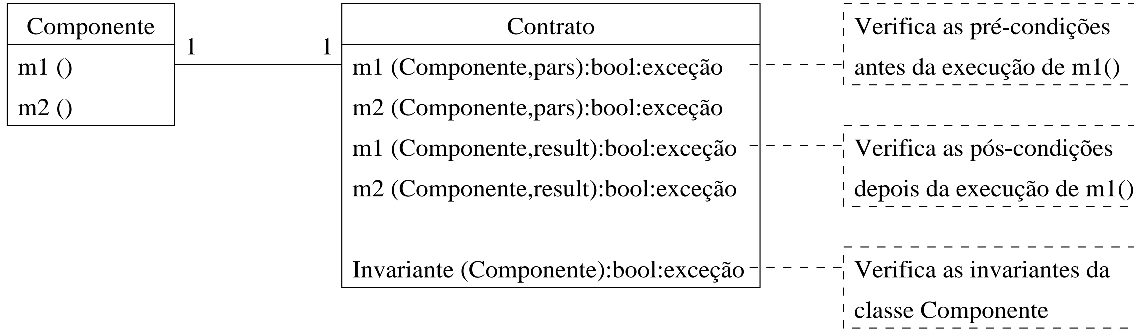

Figura 4: Diagrama de classes para o padrão Contrato

terface do modelo do componente ideal (figura 2). Os métodos que testam as pós-condições retornam exceções que correspondem às exceções de falha no modelo do componente ideal, visto 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 da classe também retorna exceções de falha, pois se as condições sobre o estado interno do componente são violadas, o componente não pode prover os seus serviços corretamente. A classificação das exceções deve ser feita de forma sistemática, e este problema será discutido com mais detalhes em outra decisão de projeto (seção 4.4).

A classe do componente ideal 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:

- Pré-condições: as pré-condições estabelecidas no contrato da superclasse podem ser **relaxadas** no subcontrato estabelecido para a subclasse. O subcontrato pode, portanto sobrecarregar os métodos que implementam as pré-condições, podendo diminuir restrições impostas sobre os parâmetro ou estado interno do objeto, ou até mesmo eliminá-las.

- Pós-condições: as pós-condições estabelecidas no contrato da superclasse podem ser mais **restritas** no subcontrato da subclasse. Ou seja, no mínimo, as restrições impostas sobre o resultado dos métodos da superclasse são atendidas, podendo ainda ser mais fortes. Desta forma, garante-se que o lado cliente do contrato da superclasse está sendo obedecido, seguindo-se as regras de subtipo, essenciais para a aplicação de polimorfismo. Para que isto seja garantido, os métodos do subcontrato que implementam as pós-condições devem invocar o método de pós-condições do supercontrato, garantido-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 estado interno de uma instância da subclasse. Portanto, o método do subcontrato que implementa a invariante da subclasse deve sempre invocar o método de invariante no seu supercontrato, podendo acrescentar novas condições.

Conseqüências: A classe Contrato faz o mapeamento da especificação de um contrato para o projeto das classes do componente ideal de uma forma mais explícita, onde todas as condições estabelecidas no contrato são implementadas explicitamente na forma de métodos. Além disto, a separação da implementação das condições do contrato do componente tolerante a falhas ideal da implementação da funcionalidade dos seus serviços torna a implementação destes serviços menos complexa, sem a necessidade de testes excessivos para a verificação de erros. A definição de uma hierarquia de contratos, paralela à hierarquia de componentes, torna mais fácil a implementação direta das regras impostas para o subcontrato, que garantem a consistência de subtipo. A solução 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 pré-condições e a restrição das pós-condições dependem da implementação destas condições.

Esta decisão de projeto propõe a definição de uma classe **Contrato** como solução para o mapeamento entre o contrato definido na especificação e o projeto e implementação de classes do componente ideal. Entretanto, surge um outro problema que se refere à localização do código que irá invocar os métodos de verificação de contrato para cada serviço do componente ideal. A próxima decisão de projeto sugere uma solução para este problema, utilizando a estrutura do padrão de arquitetura reflexão [3], para que esta detecção seja feita de uma forma transparente para o componente ideal.

4.3 Defina uma classe MetaContrato que detecta violações de contrato de forma transparente

Problema: A decisão de projeto anterior propõe a definição da classe **Contrato** para a implementação explícita das condições impostas em um contrato que estabelece os serviços confiáveis de um componente ideal, mas é preciso definir onde estes testes serão invocados. A detecção de erros é feita geralmente no código que implementa os serviços dos componentes. Considerando-se que foi definida uma classe **Contrato** que implementa as pré e pós-condições e invariantes da classe, é necessário definir onde estas verificações devem ser invocadas para garantir que os serviços sejam confiáveis. O padrão Error Detection [9] propõe uma solução para a localização dos detectores de erros de uma forma mais organizada e sistemática, testando pré-condições no início do método e pós-condições e as invariantes da classe no final do método. Mas esta solução ainda coloca as chamadas aos detectores na implementação de cada serviço. Idealmente, a implementação dos serviços que se referem às atividades normais do componente ideal não deveria se preocupar nem com a detecção de erros nem com o tratamento dos mesmos, deixando mais explícita a separação das atividades normais das atividades que se referem aos mecanismos de tolerância a falhas.

Solução: Para obter a separação entre a implementação dos serviços funcionais do componente, das atividades que garantem que estes serviços sejam confiáveis, utilize o padrão de arquitetura reflexão [3]. A solução propõe a definição de uma classe no meta-nível, chamada **MetaContrato**, responsável pela invocação dos métodos que verificam as pré e

pós-condições e invariantes da classe, de uma forma transparente para o componente ideal definido no nível base. Para isto, utiliza o mecanismo de interceptação e materialização de mensagens provido pelo protocolo de meta-objetos. A classe **MetaContrato** (figura 5) implementa dois métodos básicos: um que inspeciona uma operação interceptada antes que esta seja executada e invoca os testes de pré-condições (por exemplo, Trata(Operação)); e outra que inspeciona o resultado da operação, e invoca os testes de pós-condições e invariante da classe (por exemplo, Trata(Resultado)). Com esta solução, o componente ideal não precisa referenciar diretamente o seu respectivo contrato, nem fazer invocações explícitas aos métodos que fazem os testes.

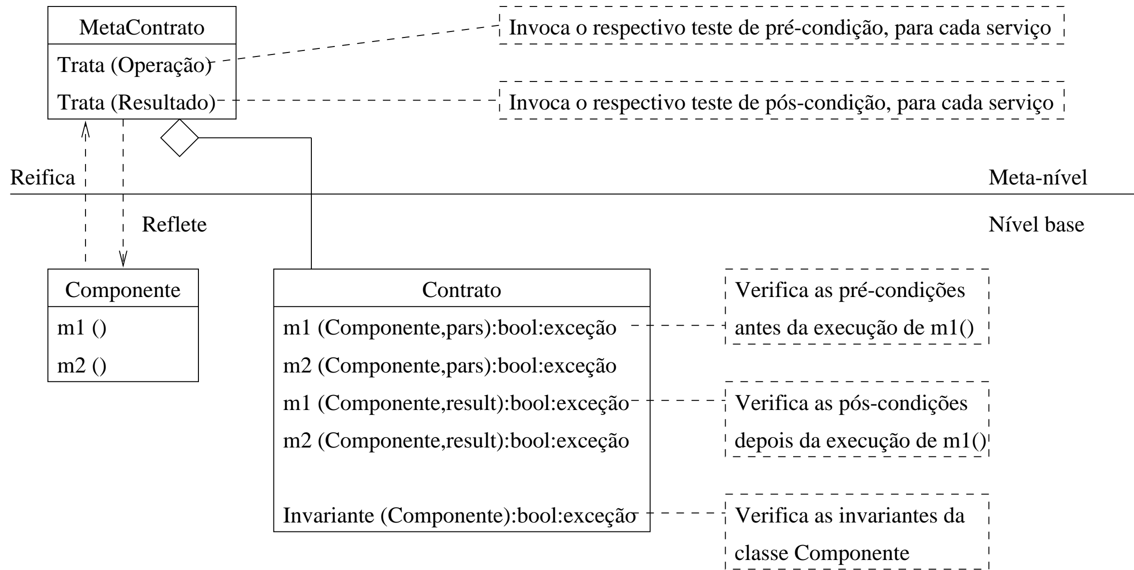

Figura 5: Diagrama de classes para o padrão MetaContrato

Conseqüências: O uso de reflexão computacional e dos mecanismos de interceptação e materialização de mensagens torna a implementação do contrato transparente para a classe do componente ideal. O **MetaContrato** funciona como uma extensão às linguagens de programação que não implementam os testes de invariante da classe, pré e pós-condições embutidos na própria linguagem, dando a ilusão de que estes testes são feitos automaticamente. Ou seja, o código dos serviços do componente não precisam incluir nenhuma chamada aos métodos da classe contrato. Isto facilita também as extensões ou modificações quanto às condições do contrato do componente ideal, visto que um meta-objeto, instância de **MetaContrato** pode ser associado ao componente ideal dinamicamente, e pode ser alterado separadamente, sem afetar a classe do componente ideal. Entretanto, com a associação dinâmica entre estes meta-objetos, instâncias de **MetaContrato** e componentes, pode-se também fazer associações indesejadas, por exemplo, associando-se uma instância de um componente ideal a um meta-objeto que não implementa o contrato da sua classe.

4.4 Defina uma hierarquia de exceções que permita a sinalização e o tratamento de exceções de forma sistemática e controlada

Problema: De acordo com o modelo de componente tolerante a falhas ideal, todos os seus serviços podem retornar uma resposta normal ou uma resposta anormal. A resposta anormal corresponde a uma exceção que deve ser retornada na cadeia de chamada dos métodos. A sinalização de exceções e o seu respectivo tratamento devem ser feitos de uma forma sistemática por todos os componentes críticos do sistema, e a especificação e classificação das exceções devem ser feitas o mais cedo possível, nas primeiras fases do projeto. Além disto, no contexto de tolerância a falhas, uma exceção possui algumas classificações que devem ser consistentemente utilizadas por todos os componentes ideais do sistema.

Considerando-se o paradigma de objetos, existem também alguns problemas relacionados com a determinação de todas as exceções levantadas pelos métodos de um objeto. Uma característica importante do sistema orientado a objetos é a herança, que permite a reutilização do projeto e implementação de classes, que evoluem desde a análise até a implementação. Desta forma, uma classe pode ser especializada e a funcionalidade dos seus métodos podem ser estendidas. De acordo com o conceito de subtipo, um método estendido não pode retornar um resultado que não atenda às pós-condições do método da superclasse. Como as exceções fazem parte da interface pública do método, toda nova exceção levantada deve ser um subtipo de alguma exceção da superclasse. Com isto a subclasse fica restrita a um subconjunto de exceções que pode não corresponder a nenhum dos novos tipos de exceções que devem ser levantados devido a extensão da nova funcionalidade. É necessário portanto, estabelecer uma hierarquia de exceções genérica que possa ser utilizada consistentemente por superclasses e subclasses, de forma que seja mantido o aspecto de subtipo das exceções, e ao mesmo tempo, não se restrinja a sinalização de novas exceções pelos métodos das subclasses.

Solução: De acordo com o modelo de componente ideal, existem dois tipos de respostas excepcionais que podem ser retornadas por um serviço: exceção de interface, que correspondem às falhas nos parâmetros de entrada do método, e exceções de falhas, que indicam que o componente não pôde prover os seus serviços normalmente. Considerando-se também as falhas de violação de contrato, é possível obter uma classificação genérica de todas as possíveis exceções que podem ser retornadas pelos serviços de um componente. Esta classificação pode ser feita nas primeiras fases de desenvolvimento de software, e podem ser especializadas nas fases subseqüentes, definindo-se tipos mais específicos. Todos os serviços devem prever a sinalização de, no mínimo, todos os tipos de exceções mais básicos, assim como os componentes clientes devem, no mínimo, prever o tratamento para estas exceções.

Utilizando-se um modelo de exceções que as define como classes, obtém-se a seguinte hierarquia de exceções:

Todas as exceções são derivadas de uma classe básica **Exception**, que generaliza todas as classes de exceções que podem ser sinalizadas e propagadas por um componente. As exceções foram classificadas inicialmente em duas exceções mais genéricas:

- **PreConditionException**: generaliza todas as exceções que podem ser levantadas

devido a uma violação de pré-condição do contrato, e que são verificadas antes do serviço ser executado. As exceções de pré-condições podem ser especializadas em: **InterfaceException**, que corresponde às violações nas condições e restrições sobre os parâmetros de entrada do método e **InternalStateException**, que correspondem às pré-condições impostas sobre o estado interno do objeto, antes que ele possa realizar um serviço.

- **FailureException**: de acordo com o modelo de componente ideal, uma exceção de falha corresponde a qualquer resposta anormal que pode ser retornada por um componente que indica que este não conseguiu prover os seus serviços normalmente. As exceções de falha podem ser especializadas em: **PosConditionException**, que indica que as pós-condições impostas no lado cliente do contrato não foram satisfeitas; **InvariantException**, que indica que a invariante da classe foi violada, e portanto, o componente não pôde realizar o serviço normalmente; e **ServiceFailureException**, que corresponde a qualquer outro tipo de falha que pode ocorrer durante a execução de um serviço, e que o componente não consegue tratar.

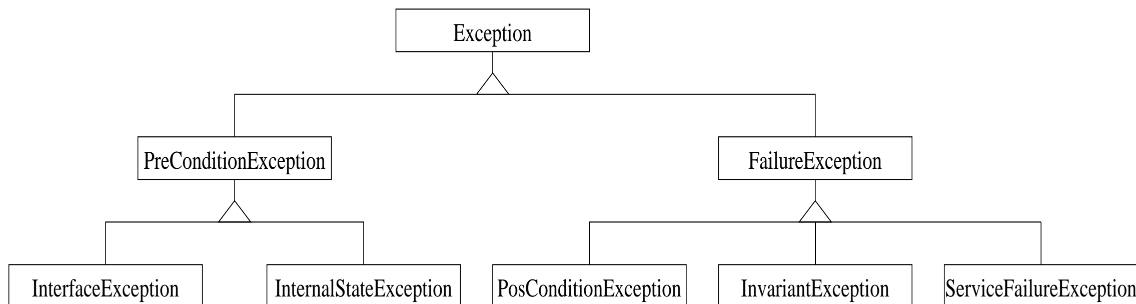

Figura 6: Hierarquia de exceções genérica

A hierarquia de exceções (Figura 6) deve ser considerada na interface de todos os serviços dos componentes ideais, desde a análise até a implementação. Pode-se iniciar a análise do componente ideal considerando-se apenas os cinco tipos mais básicos de exceções que são: **InterfaceException, InternalStateException, PosConditionException, InvariantException, e ServiceFailureException**. As fases de projeto e implementação devem especializá-las em classes de exceções mais significativas, que ajudam a determinar mais especificamente o tipo de falha que ocorreu dentro de um contexto. O que é necessário para que os mecanismos de tratamento de exceções consiga realizar adequadamente a recuperação dos erros. Através da especialização progressiva e controlada das exceções, é possível estender a funcionalidade dos métodos de uma classe sem a violação do conceito de subtipo, pois qualquer exceção nova pode ser classificada como um subtipo das cinco exceções genéricas.

Conseqüências:

- A definição de uma hierarquia genérica que classifica todos os tipos possíveis de exceções que representam as respostas anormais de um componente ideal, torna a sinalização e o tratamento de exceções mais sistemático e homogêneo em todos as classes dos componentes ideais.
- A consideração de todas as possíveis falhas na interface dos serviços dos componentes, desde análise até a implementação, diminui os riscos de defeitos no sistema devido à omissão dos casos excepcionais.
- A hierarquia de exceções permite que as subclasses definam novas exceções sem que estas violem o contrato da superclasse, sinalizando apenas exceções derivadas daquelas que foram previamente definidas pelas superclasses. Ao mesmo tempo, as subclasses não ficam restritas a um conjunto de exceções limitadas, o que ocorreria se a hierarquia de exceções não cobrisse todos os tipos de repostas excepcionais possíveis.

4.5 Separe transparentemente as atividades normais das atividades anormais de um componente

Problema: De acordo com o modelo de componente tolerante a falhas ideal, a parte anormal do componente implementa o tratamento de exceções, onde o componente tenta recuperar-se das falhas que causaram as exceções através dos seus respectivos tratadores de exceções. Idealment, o código que implementa os tratadores de exceções deveria estar separado do código que implementa a sua parte normal, pois o tratamento de exceções representa um desvio no fluxo de execução normal do programa. Este desvio inicia-se com uma busca por um tratador associado à exceção na cadeia de chamada do método. Quando uma exceção é tratada, o fluxo de controle do programa volta à sua execução normal, que corresponde à parte normal do componente ideal.

Geralmente, os tratadores de exceção são definidos no código dos métodos onde as exceções são levantadas ou dos métodos que fazem parte da cadeia de chamada do sinalizador da exceção. No contexto de tolerância a falhas, os tratadores implementam algum mecanismo de recuperação de erros, e qualquer alteração nestes mecanismos causa uma alteração no código que implementa os requisitos funcionais de um componente. Idealmente, todo o mecanismo de tolerância a falhas deveria ser introduzido no sistema de uma forma transparente e não intrusiva, separadamente da parte que implementa os requisitos funcionais.

Na maioria dos mecanismos de tratamento de exceções, a propagação de exceções ocorre na cadeia de chamada de um método. Isto cria um acoplamento entre as classes destes objetos, o que pode dificultar a manutenção e extensão destas classes. Por exemplo, se um componente (cliente) utiliza um serviço de um outro componente (servidor) e este último é estendido para adaptar-se a novos requerimentos, sinalizando alguma exceção mais especializada, os tratadores de exceções do componente cliente devem ser especializados para tratar de uma forma mais adequada a nova exceção. Utilizando a hierarquia de exceções da decisão de projeto 4 (seção 4.4) pode-se garantir que a nova exceção pertence a uma categoria de exceções que era prevista pelo componente cliente, mas este tratamento pode

não ser o mais adequado e específico para o novo subtipo de exceção. Com a implementação dos tratadores de exceções no código que implementa a parte normal do componente, uma extensão nos tratadores de exceção poderia implicar em uma extensão da classe do componente para a redefinição dos métodos que implementam os tratadores de exceção mais especializados.

Solução: Para tornar explícita a separação dos códigos que implementam a parte normal e e a parte anormal de um componente tolerante a falhas ideal, os projetistas das aplicações deveriam estruturar as suas aplicações em duas hierarquias de classes ortogonais (figura 7). Uma hierarquia de classes normais que implementa as atividades normais dos componentes da aplicação e uma hierarquia de classes excepcionais que implementa os tratadores das exceções (parte anormal dos componentes da aplicação). A hierarquia de classes excepcionais permite que subclasses excepcionais herdem tratadores de suas superclasses o que proporciona a reutilização de código anormal.

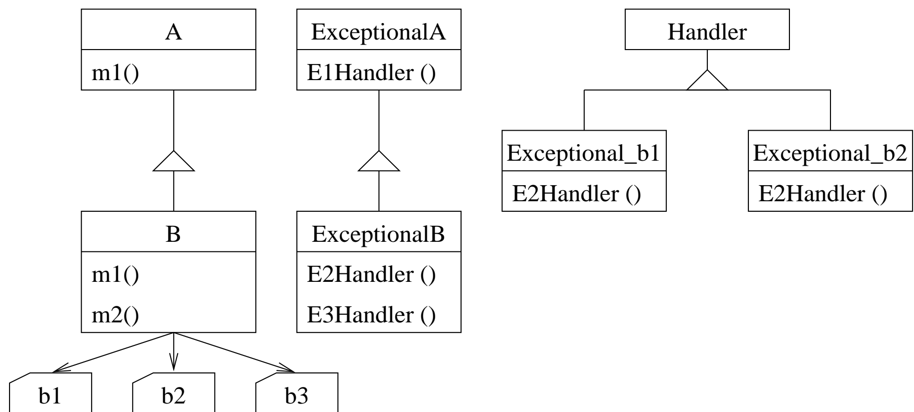

Figura 7: Hierarquia de classes normais e excepcionais

Podemos associar tratadores a classes ou a objetos individualmente. No primeiro caso, cada método das classes excepcionais são tratadores para exceções que deveriam ser tratadas pelos métodos das classes normais. Na figura 7, as classes **ExceptionalA** e **ExceptionalB** com tratadores associados às classes **A** e **B** compõem uma hierarquia de classes excepcionais que é ortogonal a hierarquia de classes normais da aplicação. Os métodos da classe **ExceptionalA** são tratadores para as exceções que deveriam ser tratadas nos métodos da classe **A**. Os tratadores para as exceções que deveriam ser tratadas nos métodos da classe **B** ou são métodos da classe **ExceptionalB** ou são herdadas das superclasses desta. Por exemplo, o tratador para a exceção **E1** (**E1Handler ()**) é herdado de **ExceptionalA**.

No segundo caso, associamos tratadores a objetos individualmente. Por exemplo, o objeto **b1**, instância da classe **B**, pode ser associado a tratadores distintos dos tratadores associados ao objeto **b2** que também é uma instância da classe **B**. Na figura 7, métodos da classe **Exceptional_b1** são tratadores para as exceções que deveriam ser tratadas no objeto **b1**. Além disso, é possível uma única classe excepcional ser associada para um grupo de objetos. Por exemplo, tratadores associados ao objeto **b3**, instância da classe **B**, podem ser

os mesmos tratadores associado ao objeto **b2**, isto é, estes objetos estão associados a uma mesma classe excepcional (**Exceptional_b2**). Desta forma, **b2** e **b3** tem comportamentos anormais idênticos mas diferentes do comportamento anormal de **b1**.

Para obter a transparente associação entre a hierarquia de classes normais da hierarquia de classes excepcionais, utilizamos o padrão de arquitetura reflexão [3]. Exceções levantadas pelos componentes são interceptadas pelo meta-nível e a procura pelo adequado tratador para aquela exceção é efetuado por meta-objetos de forma transparente para o programador da aplicação. A figura 8 apresenta a classe **MetaHandler** que é responsável pelo gerenciamento das atividades de tratamento de exceções. Baseado nas operações e nos seus resultados, instâncias de **MetaHandler** executam as seguintes atividades: (i) procura do tratador associado àquela exceção; (ii) invocação deste tratador; (iii) retorno ao fluxo normal da aplicação. Nesta figura o objeto **x1** está associado a duas classes excepcionais: uma que implementa os tratadores específicos a este objeto (**Exceptional_x1**) e uma que implementa os tratadores comuns a todas as instâncias da classe **X** (**ExceptionalX**).

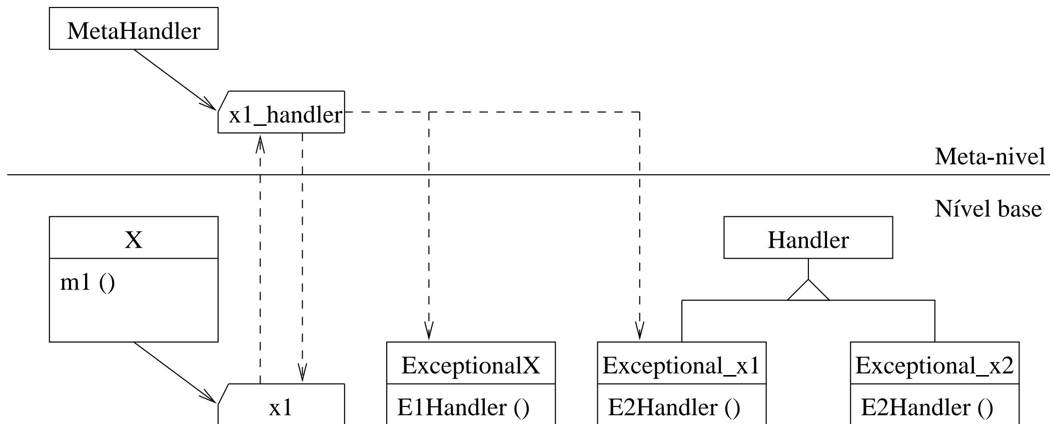

Figura 8: Associação entre objetos e seus tratadores de exceções

Um mecanismo de tratamento de exceções que dá suporte a separação entre as hierarquias normais e excepcionais de uma aplicação e a associação de tratadores a classes e objetos foi recentemente projetado [5] e acreditamos que essa decisão de projeto pode ser facilmente implementada utilizando este mecanismo de tratamento de exceções.

Conseqüências:

- A implementação de todo o tratamento de exceções no meta-nível torna a implementação dos métodos mais limpa, contendo apenas o fluxo de programa que implementa as atividades normais dos componentes. Isto torna mais explícita a separação das atividades normais das anormais do componente tolerante a falhas ideal, facilitando o entendimento e a manutenção das classes.

- Esta solução substitui os mecanismos de tratamento de exceções implementados pela linguagem de programação, implementando através dos meta-objetos e do mecanismos de interceptação de mensagens todo o mecanismo de busca por um tratador e

propagação da exceção. Esta solução fornece mais flexibilidade na implementação destes mecanismos, mas pode introduzir erros de programação, desde que o compilador não será mais responsável por testar se existe um tratador associado a uma exceção, ou se uma exceção já tratada continua sendo propagada, etc.

4.6 Para componentes que não foram previamente projetados de acordo com o modelo de componente tolerante a falhas ideal, implemente todo o mecanismo de exceções através de meta-objetos

Problema: Vários sistemas são desenvolvidos sem a preocupação com os mecanismos de tolerância a falhas, e portanto, seus componentes não são projetados de acordo com o modelo de componente tolerante a falhas ideal. Nestes sistemas, a interface dos serviços de um componente não prevê a ocorrência de respostas excepcionais (exceções) em decorrência de falhas na execução de um serviço, e conseqüentemente não definem os mecanismos de tratamento de exceções que implementam a recuperação destas falhas. Entretanto, pode ser necessário estender o sistema para que este se adapte a novos requisitos de robustez e confiabilidade dos seus serviços. Esta extensão não é uma tarefa trivial, visto que a inclusão dos mecanismos de tolerância a falhas, utilizando exceções e tratamento de exceções, pode afetar a interface e a implementação de todos os serviços dos componentes do sistema. Idealmente, a extensão do sistema aos novos requisitos de tolerância a falhas não deve afetar os componentes que implementam os requisitos funcionais do sistema.

Solução: A maioria das decisões de projeto anteriores utilizam o padrão de arquitetura reflexão [3] para o projeto dos componentes do sistema de acordo com o modelo de componente tolerante a falhas ideal. A arquitetura reflexiva proporciona uma separação de tarefas, de forma que todas as atividades anormais do componente, que implementam os mecanismos de tratamento de exceções necessários para a implementação de tolerância a falhas, sejam executadas de uma forma transparente e não intrusiva no meta-nível (seção 4.5). Além disto, a implementação dos componentes no nível base não fica **poluída** com testes de detecção de erros e violações de contrato, que também são implementados transparentemente pelos meta-objetos (seção 4.3). Todas estes padrões podem também ser aplicados para estender componentes de sistemas não-robustos, que foram projetados sem atender ao modelo de componente tolerante a falhas ideal. O relacionamento entre os meta-objetos e os objetos do nível base é feito de uma forma transparente e não intrusiva, através dos mecanismos de interceptação e materialização de mensagens, e desta forma, é possível tornar um componente do sistema em um componente tolerante a falhas ideal, sem que a implementação de seus serviços seja afetada. Além disto, a associação entre um meta-objeto e um objeto do nível base pode ser feita dinamicamente, dependendo do protocolo de meta-objetos utilizado, e assim, esta adaptação do componente pode ser feita em tempo de execução.

Um componente tolerante a falhas ideal possui uma interface bem definida, que sinaliza todas as possíveis respostas excepcionais que podem ser retornadas devido a falhas na execução de um serviço. Entretanto, um componente que não foi implementado como um componente tolerante a falhas ideal não tem definido em sua interface a sinalização destas exceções, como estabelece a decisão de projeto 4 (seção 4.4). A maioria das linguagens que

implementa os mecanismos de exceções fazem a verificação da especificação das exceções, não permitindo que exceções não previstas na interface de um método seja propagada pela cadeia de execução. Assim, é preciso **burlar** esta verificação, para que exceções não definidas na interface possam ser levantadas pelos meta-objetos capazes de detectar erros (meta-objeto MetaContrato), e sejam propagadas pela cadeia de execução até que sejam tratadas por objetos (instâncias de classes pertencentes a hierarquia de classes excepcionais) que implementam a parte anormal dos componentes. Com esta solução, toda a sinalização e tratamento de exceções é gerenciado pelo meta-nível, sem nenhuma alteração na interface ou na implementação dos serviços dos componentes do nível base.

A aplicação desta solução irá depender da linguagem de programação utilizada. Em Java, por exemplo, é possível definir classes de exceções derivadas da classe **RuntimeException**, as quais não precisam está declaradas na interface do método, e não precisam ser obrigatoriamente tratadas. A classe **RuntimeException** pode ser a classe base para a hierarquia de exceções discutida na seção 4.4 para que todos os tipos de exceções possam ser sinalizados e propagados.

Conseqüências:

- Aplicando-se esta solução, um componente não-robusto pode se tornar um componente tolerante a falhas ideal, sem que a implementação e a interface dos seus serviços sejam alteradas. Desta forma, um sistema pode ser totalmente adaptado para atender a novos requisitos não-funcionais de robustez e confiabilidade, de forma transparente e não intrusiva para os componentes que implementam a parte funcional da aplicação.

- Uma desvantagem é que esta solução não utiliza o apoio da linguagem de programação para a verificação da especificação das exceções na interface dos serviços, a propagação e o tratamento de exceções, o que torna o sistema mais susceptível a erros de programação. Um exemplo de erro comum é a omissão de um tratador para um tipo de exceção, o que pode levar o sistema a um estado errôneo, provocando um defeito inesperado no sistema. Erros deste tipo seria verificado pelo compilador, se estivessem sendo utilizados os mecanismos de exceções da linguagem, onde as exceções deve fazer parte da interface de um método, obrigando assim os clientes de um método a tratarem ou propagarem a exceção.

5 Exemplo do uso do padrões

Para ilustrar o uso desta metodologia na construção de componentes tolerantes a falhas, o projeto de um contador será descrito. Este contador têm um limite inferior (*lower*) e um superior (*upper*). A interface desta classe provê dois métodos para incrementar o contador: $inc()$ que incrementa o contador em uma posição e $add(n)$ que adiciona n posições ao contador. As operações de incrementar ou adicionar apenas será executada se o valor resultante é menor que o limite superior (*upper*) e que a invariante da classe ($lower <= count <= upper$) não é desrespeitada. A figura 9 ilustra a arquitetura de meta-nível resultante da implementação deste componente.

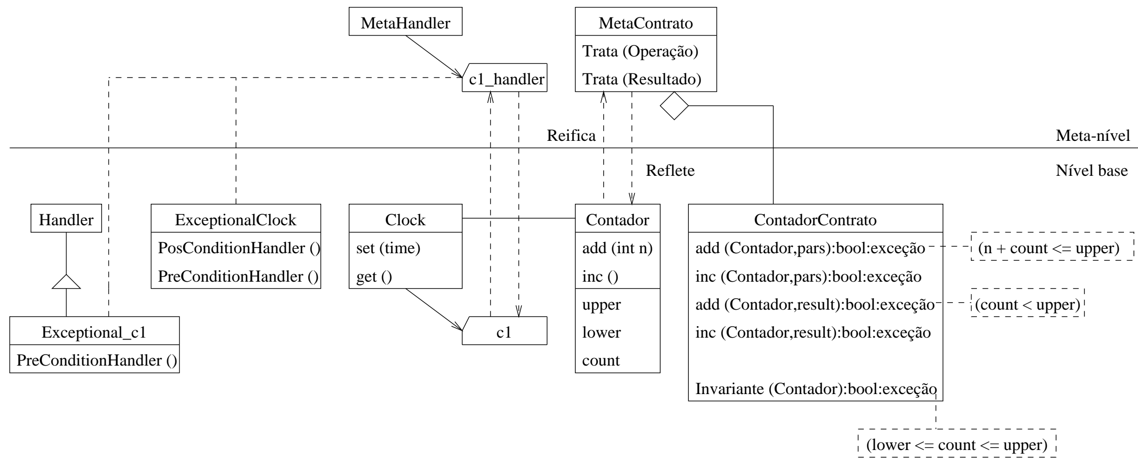

Figura 9: Exemplo da implementação de um componente tolerante a falhas ideal

Nesta figura, a classe **Clock** utiliza os serviços providos pelo contador. Durante a execução dos serviços do contador, exceções podem ser levantadas e estas devem ser tratadas pelas instâncias da classe **Clock**. As invariantes, as pré e as pós-condições da classe **Contador** são representadas por métodos da classe **Contrato**. Instâncias da classe **MetaContrato** associados a instâncias da classe **Contador** são responsáveis por verificar o cumprimento do contrato. Se estas condições não são satisfeitas exceções são levantadas e os tratadores associados a estas exceções são métodos da classes **ExceptionalClock** e **Exceptional_c1**.

6 Considerações finais

Propomos neste trabalho um conjunto de padrões para o desenvolvimento de aplicações tolerantes a falhas. Os padrões representam decisões de projeto que devem ser tomadas no desenvolvimento de sistemas orientados a objetos tolerantes a falhas para a obtenção de um projeto estruturado, onde a introdução dos mecanismos de tolerância a falhas é feita de forma controlada e consistente, em todas as fases do ciclo de desenvolvimento do software. Os padrões são fortemente relacionados, e cada um resolve um problema dentro de um contexto particular em cada fase de desenvolvimento dos componentes da aplicação. Estes padrões baseiam-se no modelo de componente tolerante a falhas ideal que possui uma separação clara entre o seu comportamento normal e anormal. Contratos são utilizados para especificar o comportamento normal e anormal de cada componente e reflexão computacional para tornar explícita esta separação entre o comportamento normal e anormal de cada componente.

Os padrões descritos neste trabalho não leva em consideração a construção de sistemas concorrentes tolerantes a falhas. Em muitos casos, diversas atividades concorrentes estão trabalhando juntas, isto é, cooperando para resolver um determinado problema. Em outros casos, as atividades são independentes, mas competem no acesso a recursos comuns.

O conceito de ação atômica coordenada [10] é a primeira tentativa de integrar transações atômicas [2] e conversações [8] em um mesmo arcabouço que lida com ambos tipos de concorrência (cooperação e competição). Ações atômicas coordenadas provêm 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. Um framework reflexivo para a implementação deste conceito foi recentemente projetado [1]. No entanto, um padrão que captura completamente este conceito ainda está sendo projetado.

Referências

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

[2] P.A. Bernstein, V. Hadzilacos e N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addisson-Wesley Publishing Company, 1987.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad e M. Stal. *A System of Patterns: Patterns-Oriented Software.* John Wiley & Sons, 1996.

[4] M. de Champlain. The Contract Pattern. *The 1997 Pattern Languages of Programs Conference (PLoP97)*, Monticello, Illinois, EUA, Setembro 1997.

[5] A.F. Garcia, D.M. Beder e C.M.F. Rubira. An Exception Handling Mechanism for Developing Dependable Object-Oriented Software Based on a Meta-level Approach. *A ser publicado nos anais do X International Symposium on Software Reliability Engineering (ISSRE'99)*, Florida, EUA, Novembro 1999.

[6] P.A. Lee e T. Anderson. *Fault Tolerance: Principles and Practice.* Springer-Verlag, 2a. edição, 1990.

[7] B. Meyer. *Object-Oriented Software Construction.* New York: Prentice-Hall, 1988.

[8] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, 1975.

[9] Klaus Renzel. Error Detection. *Second European Conference on Pattern Languages of Programming and Computing (EuroPLOP'97)*, Kloster Irsee, Alemanha, Julho 1997.

[10] J. Xu, B. Randell, A. Romanovsky, C.M.F. Rubira, R. Stroud e Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. *25th International Symposium on Fault-Tolerant Computing (FTCS-25)*, páginas 499–509, Pasadena, California, EUA, Junho 1995.