Padrão State Reflexivo: Refinamento do Padrão de Projeto State para uma Arquitetura Reflexiva

Luciane Lamour Ferreira *Cecília Mary Fischer Rubira*

Universidade Estadual de Campinas - UNICAMP
Instituto de Computação
e-mail: {972311,cmrubira}@dcc.unicamp.br
Campinas, SP 13083-970
Caixa Postal 6176, Fax: (019)788-5847

Resumo

Padrões constituem boas soluções de projeto para problemas recorrentes dentro de um contexto particular. À medida que os padrões são aplicados, aumenta-se o entendimento sobre os mesmos, e as soluções apresentadas por eles podem evoluir, tornando-os cada vez mais maduros. Este artigo apresenta um refinamento para o padrão de projeto *State* e discute vários problemas associados à sua implementação. Para a aplicação do padrão *State*, são necessárias decisões de projeto que envolvem aspectos de controle relacionados ao comportamento dinâmico de um objeto, como por exemplo, a definição e o controle das transições de estado. O padrão proposto, denominado padrão *State* Reflexivo, utiliza uma arquitetura reflexiva para a implementação destes aspectos de controle no meta-nível, separando-os dos aspectos funcionais implementados no nível base. São definidas também algumas variações do padrão *State Reflexivo* para o domínio de tolerância a falhas, dando origem a um sistema de padrões que auxilia o desenvolvimento de software tolerante a falhas.
Palavras-chave: Padrões de Projeto, Reflexão Computacional, Arquitetura Reflexiva, Framework Orientado a Objetos e Tolerância a Falhas.

Abstract

Patterns are good design solutions for recurring problems that occur in a particular context. As patterns are applied, their understanding grows, and their design solutions can evolve and, consequently, can become more mature. This paper presents a refinement of the State pattern and discuss some problems related to its implementation. In order to apply the State pattern, it is necessary to take design decisions related to the control aspects of the dynamic behavior of the object, such as definition and control of state transitions. The proposed pattern, called Reflective State pattern, applies a reflective architecture to implement these control aspects in a meta level, separating them from the functional aspects which are implemented in the base level. Some variations of the Reflective State pattern to the fault tolerance domain are also proposed, originating a system of patterns that help the development of fault-tolerant software.
Key-words: Design Patterns, Computational Reflection, Reflective Architecture, Object-Oriented Frameworks, Fault Tolerance.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

1. Introdução

O desenvolvimento de software orientado a objetos tem sido amplamente auxiliado pelo uso de padrões de projeto. A experiência de desenvolvedores de software é documentada através destes padrões, que registram boas soluções de projetos para problemas recorrentes dentro de um contexto particular, permitindo a reutilização destas soluções. Os padrões podem existir em diferentes níveis de abstração, auxiliando nas diversas fases do ciclo de desenvolvimento de software, podendo ainda ser combinados para formar soluções mais complexas. Atualmente, existe uma grande quantidade de padrões documentados, agrupados em catálogos[GHJV95], sistemas de padrões[BMRS+96] ou linguagens de padrões [DA96]. À medida que os padrões são aplicados, aumenta-se o entendimento sobre os mesmos, e as soluções de projeto apresentadas por eles podem evoluir, tornando-os cada vez mais maduros. Além disto, a aplicação de um padrão pode dar origem a outros problemas que também podem ser resolvidos por outros padrões.

O padrão de projeto *State* [GHJV95] é um padrão bastante conhecido, e seu objetivo é permitir que um objeto altere o seu comportamento quando seu estado interno muda. Para isto, ele define uma hierarquia de estados paralela à hierarquia de classes do objeto, e utiliza o conceito de delegação, onde o objeto da aplicação delega operações dependentes de estado para o seu objeto de estado corrente. Na implementação do padrão *State*, são necessárias decisões de projeto que envolvem aspectos de controle relacionados ao comportamento dinâmico de um objeto, como por exemplo, a definição e o controle das transições de estado. Se o objeto possui um comportamento dinâmico complexo, a implementação dos aspectos de controle, em geral, não é trivial. Os aspectos de controle ficam geralmente "misturados" à implementação do objeto de contexto da aplicação ou à implementação dos objetos de estado. No entanto, as duas abordagens aumenta o acoplamente entre as classes, dificulta a extensão tanto da hierarquia de classes do objeto de contexto quanto da hierarquia de estados, dificultando também a reutilização das duas hierarquias separadamente.

Este artigo mostra um refinamento do padrão *State* para uma arquitetura reflexiva, denominado padrão *State Reflexivo*. O principal objetivo deste padrão é propor uma solução para os problemas de implementação do padrão *State*, enfatizando a separação dos aspectos de controle do padrão, que são definidos no meta-nível, dos aspectos estritamente funcionais, que são definidos no nível base. É mostrada também uma variação do padrão *State Reflexivo* para o domínio de tolerância a falhas, e outras variações que consideram técnicas de tolerância a falhas específicas. O conjunto destes padrões formam um sistema de padrões que auxiliam no desenvolvimento de software tolerante a falhas.

O restante deste artigo está organizado da seguinte forma. A Seção 2 discute alguns conceitos relacionados a Padrões e Reflexão Computacional. A Seção 3 apresenta o padrão *State* e alguns conceitos relacionados ao comportamento dinâmico de um objeto. A Seção 4 apresenta o padrão *State Reflexivo*. A Seção 5 mostra uma variação do padrão *State Reflexivo* para o domínio de tolerância a falhas e o sistema de padrões. A Seção 6 apresenta as conclusões deste trabalho.

2. Padrões de Projeto e Reflexão Computacional

2.1 Padrões de Projeto

Padrões capturam experiências coletivas comprovadas no desenvolvimento de software e ajudam a promover boas práticas de projeto. Cada padrão consiste de uma regra de três partes [Ale79]: (i) um certo contexto, (ii) um problema e (iii) uma solução. Os padrões

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

possuem diferentes níveis de abstração e podem auxiliar em todas as fases do ciclo de desenvolvimento de software, como foi dito anteriormente. Segundo Buschmman[BMRS+96], os padrões podem ser agrupados em três categorias:

- Padrões de Arquitetura, que expressam um esquema de organização estrutural e fundamental para um sistema de software. Eles provêem um conjunto de subsistemas pré-definidos, especificando seus relacionamentos, e estabelecendo regras e diretrizes para a organização dos relacionamentos entre eles.
- Padrões de Projeto, que provêem um esquema para refinar os subsistemas ou componentes de um sistema de software. Os padrões de projeto descrevem uma estrutura que constitue uma solução para um problema de projeto geral dentro de um contexto particular.
- Idiomas, que são padrões de nível mais baixo, específicos para uma determinada linguagem de programação. Eles descrevem como implementar um aspecto particular de um componente ou o relacionamento entre eles usando características de uma linguagem específica.

Os padrões não são aplicados isoladamente, pois um único padrão não é capaz de resolver todos os problemas de software, em todos os níveis de abstração. Portanto, existem relacionamentos entre os padrões que os tornam fortemente conectados. A utilização de um padrão para solucionar um problema mais genérico pode originar um outro problema que também pode ser solucionado por um padrão mais específico, ou por uma combinação de padrões. Em [BMRS+96] são apresentados três tipos de relacionamentos entre padrões: refinamento, que dá suporte na implementação de um padrão; combinação, que ajuda na composição de estruturas de projeto complexas; variações que auxiliam na seleção de um padrão mais adequado dentro de uma situação específica de projeto.

Para que se tenha uma reutilização efetiva das soluções propostas pelos padrões, eles devem ser documentados e registrados de uma forma uniforme. Existem basicamente três tipos de coleções de padrões: os catálogos de padrões[GHJV95], que apresentam um conjunto de padrões em um formato uniforme, independente de domínio, e que não possuem um relacionamento forte; os sistemas de padrões[BMRS+96], que apresentam uma coleção de padrões relacionados, que ajudam a desenvolver sistemas com propriedades particulares; e as linguagens de padrões[DA96], que formam um conjunto fechado de padrões fortemente relacionados ligados por regras bem definidas, que resolvem todos os aspectos relacionados a um domínio de problema.

2.2 Reflexão Computacional

Reflexão computacional[Mae87] [OGB98] pode ser definida como sendo a habilidade de observar e manipular o comportamento computacional de um sistema através de um processo de materialização, de forma que o sistema possa usar a representação de si mesmo para estender e adaptar suas computações. Reflexão pode ser usada para interceptar e modificar o efeito de várias operações básicas, como a criação de um objeto, a invocação de um método ou a criação de uma classe[BRL97].

O padrão de arquitetura *Reflection* [BMRS+96] divide uma aplicação em basicamente dois níveis: meta-nível e nível base. No meta-nível residem os componentes responsáveis pelas ações administrativas a serem realizadas sobre um sistema alvo localizado no nível base (nível da aplicação). Estes componentes são chamados de metaobjetos. No nível base residem os componentes que lidam com a funcionalidade básica da aplicação. A ligação entre os componentes do meta-nível com os do nível base é feita transparentemente, através de um mecanismo de interceptação e materialização de mensagens, de forma que se possa refletir sobre a computação do nível base.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

Arquiteturas reflexivas provêem um protocolo de metaobjetos (MOP) que estabelece a forma como os objetos de nível base e os de meta-nível estão relacionados e de que forma os metaobjetos podem interferir nos objetos de nível base. Uma linguagem reflexiva deve permitir a definição dos metaobjetos e a realização dos mecanismos de interceptação de mensagens e de materialização. Vários trabalhos na literatura têm relacionado o uso de arquiteturas reflexivas para a obtenção da separação dos aspectos que implementam os requisitos estritamente funcionais da aplicação, dos aspectos que implementam os requisitos não-funcionais, tais como distribuição[OB98], tolerância a falhas[BRL97], etc. Com esta abordagem, é possível construir sistemas complexos estruturados, que atendem a vários requisitos não-funcionais, e que podem ser mais facilmente adaptados, estendidos e mantidos.

3. O padrão *State* e suas várias implementações

3.1 Padrão State

O padrão de projeto *State* [GHJV95] propõe uma solução genérica para o problema de objetos que podem apresentar diversos estados, e apresentam comportamentos distintos dependendo de seu estado corrente. Para isto, ele define uma hierarquia de estados paralela à hierarquia de classes do objeto, e utiliza o conceito de delegação, onde o objeto da aplicação delega operações dependentes de estado para um objeto que representa o seu estado corrente. Em tempo de execução, o objeto da aplicação pode mudar de estado, implicando também numa mudança da implementação do seu comportamento.

Os aspectos de implementação do padrão *State*[GHJV95] discutem onde deve estar localizada a responsabilidade da definição dos estados e gerenciamento das transições, e propõem duas possívies soluções: (a) o controle das transições pode ficar centralizado no objeto do contexto da aplicação, que conhece todos os seus possíveis estados, trata os eventos que provocam as transições, e realiza a transição mudando o seu estado corrente; (b) o controle das transições pode estar "espalhado" nos objetos de estado, onde cada um conhece seus próximos estados e é o responsável por verificar os eventos e realizar as transições. A principal desvantagem da primeira solução é que o objeto de contexto tem todo o controle sobre os estados e transições, e a implementação de sua funcionalidade básica fica misturada a estes aspectos de controle. A desvantagem da segunda abordagem é o aumento das dependências entre as classes de estado (alto acoplamento), o que também dificulta a extensão e manutenção das mesmas. Vários trabalhos na literatura discutem estes problemas e apresentam também variações ou refinamentos para o padrão *State* [DA96] [OS96] [Ran95] [Pal97]. Estes padrões são discutidos na seção 4.3.

3.2 Modelagem de Comportamento Dinâmico

O comportamento dinâmico de um objeto é geralmente modelado utilizando-se diagramas de estado, que são compostos de três elementos principais: estados, eventos e transições. *Statecharts* estruturados [Har87] estendem os diagramas de transição de estado com as noções de superestados (generalização), subestados (especialização) e estados concorrentes (ortogonais). Além disto, os *statecharts* consideram também condições de guarda e ações de saída na execução de uma transição.

O padrão *State* pode ser visto como um mecanismo de implementação de *statecharts* no paradigma de objetos, onde a hierarquia de estados implementa os estados do *statechart* correspondente para a classe do objeto de contexto. Os aspectos de controle do *statechart*,

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

como as transições de estado, ficam espalhados na implementação do objeto de contexto ou dos objetos de estado, como mencionado anteriormente.

4. Definição do padrão State Reflexivo

De acordo com as classificações e definições dadas na Seção 2.1, pode-se classificar o padrão *State Reflexivo* como sendo um refinamento do padrão *State*[GHJV95], que trata de problemas relacionados à sua implementação. Este padrão segue a estrutura geral do padrão de arquitetura *Reflection* [BMRS+96]. O padrão *State Reflexivo* é um padrão de projeto genérico, que possui algumas variações para problemas de domínios mais específicos, como por exemplo, o domínio de tolerância a falhas. O conjunto de padrões formado pelo padrão de arquitetura *Reflection*, pelo padrão *State Reflexivo* e pelas variações deste último para o domínio de tolerância a falhas, formam um sistema de padrões que pode auxiliar no desenvolvimento de software tolerante a falhas, como discutido na Seção 5.

O padrão *State Reflexivo* é apresentado de acordo com o seguinte formato: nome, objetivo, contexto, problema, solução, estrutura estática e dinâmica. São discutidos também as conseqüências da sua aplicação e alguns padrões relacionados.

4.1 Padrão State Reflexivo

Objetivo

Apresentar uma solução para as decisões de projeto que devem ser feitas na implementação do padrão *State*[GHJV95]. Considera uma arquitetura reflexiva para obter a separação das tarefas de controle do padrão *State*, definindo uma representação de *statecharts* no meta-nível, que deverá controlar as transições, a definição dos estados, o estado corrente do objeto e as delegações de serviços dependentes de estado para os objetos de estado.

Contexto

Um objeto pode ter um comportamento dinâmico bem definido, possuindo vários estados e comportamentos diferentes dependendo de seu estado corrente. Para a especificação do comportamento dinâmico de uma classe de objetos pode-se utilizar diagramas de estados ou *statecharts* estruturados [Har87]. Por definição, uma máquina de estado é composta por um conjunto de estados, um conjunto de eventos e um conjunto de transições, e esta última é uma função do estado corrente e de um evento ocorrido e produz como saída um novo estado. Em se tratando de *statecharts*, pode-se ainda considerar condições de guarda que controlam transições, e ações de saída, além de aspectos de herança, composição, e concorrência de estados, acrescentando-se complexidade no controle da máquina.

Problema

Dada uma especificação do comportamento dinâmico de um objeto utilizando-se *statecharts*, a sua implementação em uma linguagem orientada a objetos não é direta. Como foi dito anteriormente, deve-se considerar o controle das transições de estado, a definição e o conhecimento de todos os possíveis estados, a verificação dos eventos que provocam transições e o teste de condições de guarda. Os principais problemas e restrições que devem ser considerados são:

- No padrão *State*[GHJV95] os aspectos de controle da máquina de estado se misturam à implementação do objeto de contexto ou dos objetos de estado. Isto aumenta a complexidade, dificulta o entendimento, e a reutilização da classe do objeto de contexto e da

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

hierarquia de estados, separadamente. O problema a ser considerado é, então, a separação de tarefas de controle das tarefas que implementam a funcionalidade da aplicação.

• Além do problema da separação de tarefas, deve-se considerar também o problema de transparência nos mecanismos de controle. Para obter a máxima reutilização dos componentes que implementam a funcionalidade da aplicação, estes componentes não devem ter nenhum conhecimento sobre os mecanismos de controle do *statechart*.

Solução

A solução proposta para os problemas descritos anteriormente é a utilização de uma arquitetura reflexiva para a obtenção da separação das tarefas de controle dos *statecharts*, que são levados para o meta-nível, das tarefas que implementam a funcionalidade específica da aplicação, que são implementadas no nível base. Utilizando-se esta arquitetura reflexiva, obtem-se transparência na execução dos aspectos de controle, através do mecanismo de interceptação de mensagens que é provido pelo protocolo de metaobjetos. Os objetos de contexto e os objetos de estado provêem seus serviços sem que tenham o conhecimento dos mecanismos de controle dos *statecharts*.

Estrutura estática

O controle dos *statecharts* é totalmente implementado no meta-nível através da abstração dos elementos que o compõem, ou seja, estados e transições, que são representados por metaobjetos. No meta-nível, são definidos: um metaobjeto que representa o controlador da execução do *statechart* (*MetaController*); uma hierarquia de metaobjetos que representam os possíveis estados, considerando estados e subestados do *statechart* (*MetaStates*); uma hieraquia de metaobjetos que representam as transições, podendo também formar uma hieraquia de transições (*MetaTransitions*). Os eventos que ativam transições são controlados pelo *MetaController* através da interceptação de mensagens provida pelo protocolo de metaobjetos. A Figura 1 apresenta o diagrama de classes que mostra as classes dos objetos do nível base e do meta-nível. As classes do nível base correspondem às mesmas classes definidas pelo padrão *State*[GHJV95]. Em termos de estrutura estática, a principal alteração feita pelo padrão *State Reflexivo* foi a retirada do relacionamento de agregação entre a hieraquia de classes do objeto de contexto com a hierarquia de classes de estado.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

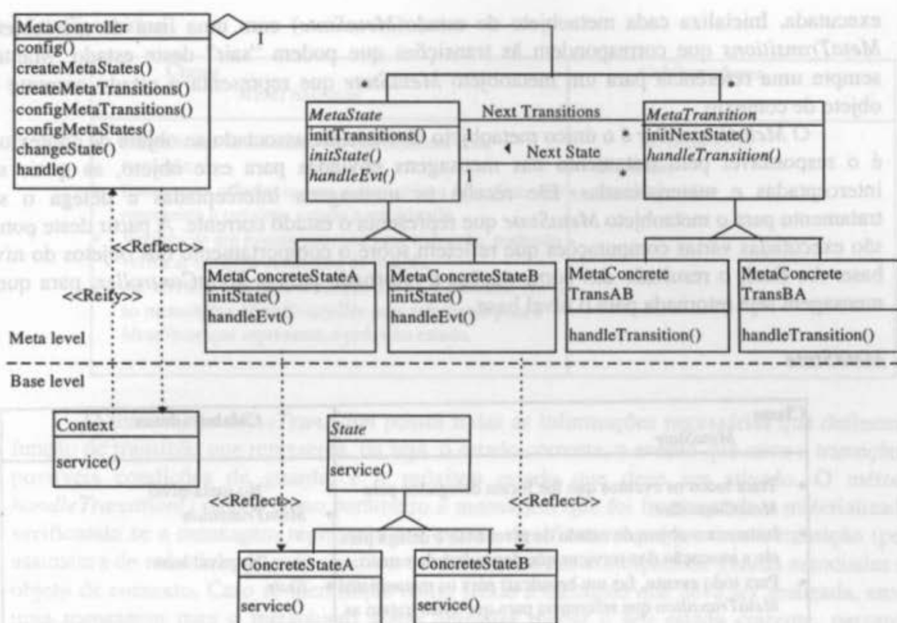

Figura 1: Diagrama de classes do padrão *State* Reflexivo usando a notação UML[BRJ97]

As responsabilidades dos elementos que formam a estrutura do padrão são descritas utilizando os cartões CRC (Classe-Responsabilidades-Colaboradores) que mostram as responsabilidades e os colaboradores de cada classe de objeto do meta-nível e do nível base.

MetaController

Classe *MetaController*	**Colaboradores**
• Configura o meta-nível, criando e inicializando os metaobjetos *MetaState* e *MetaTransition*. • Intercepta todas as mensagens enviadas para o objeto de contexto. • Controla e mantém uma referência para o metaobjeto *MetaState* que representa o estado corrente. • Delega o tratamento de todas as mensagens interceptadas para o metaobjeto *MetaState* corrente. • Efetua as transições de estado.	No meta-nível: • *MetaState* • *MetaTransition* No nível base: • *Context*

O metaobjeto *MetaController* é responsável pela configuração do meta-nível de acordo com a especificação de um *statechart* que modela o comportamento dinâmico dos objetos da classe *Context*. Para isto, instancia e inicializa os metaobjetos de estado (*MetaStates*) e os metaobjetos de transição (*MetaTransitions*) que representam os estados e as transições do *statechart*. Inicializa cada metaobjeto *MetaTransition* com uma referência para um metaobjeto *MetaState* que representa o próximo estado que deve ser ativado quando a transição for

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

executada. Inicializa cada metaobjeto de estado(*MetaState*) com uma lista de metaobjetos *MetaTransitions* que correspondem às transições que podem "sair" deste estado. Mantém sempre uma referência para um metaobjeto *MetaState* que representa o estado corrente do objeto de contexto.

O *MetaController* é o único metaobjeto diretamente associado ao objeto de contexto, e é o responsável pelo tratamento das mensagens enviadas para este objeto, as quais são interceptadas e materializadas. Ele recebe as mensagens interceptadas e delega o seu tratamento para o metaobjeto *MetaState* que representa o estado corrente. A partir deste ponto, são executadas várias computações que refletem sobre o comportamento dos objetos do nível base. Ao final, o resultado das computações é retornado para o *MetaController*, para que a mensagem seja retornada para o nível base.

MetaState

Classe *MetaState*	Colaboradores
• Trata todos os eventos que lhe foram delegados pelo *MetaController*. • Instancia o objeto de estado do nível base e delega para ele a execução dos serviços que dependem do estado. • Para todo evento, faz um broadcast para os metaobjetos *MetaTransition* que referencia para que estes tratem as transições. • Retorna para o *Metacontroller* a mensagem que lhe foi passada com os resultados do seu tratamento.	No meta-nível • *MetaTransition* No nível base • *State*

Os metaobjetos *MetaState* são responsáveis pela inicialização do tratamento das mensagens materializadas. Através do método *handleEvt()* o metaobjeto *MetaState* corrente recebe como parâmetro o evento materializado para verificar que tipo de mensagem foi solicitada para o objeto. Se o evento corresponde a um serviço dependente de estado, a mensagem é enviada para o objeto de estado corrente no nível base (que realmente implementa o serviço), e o resultado da execução é retornado para o metaobjeto *MetaState* para que este possa retorná-lo para o metaobjeto *MetaController*. Além disto, para toda mensagem interceptada, também faz o *broadcast* para a sua lista de metaobjetos *MetaTransitions* (que representam as transições que saem do estado corrente), para que estes verifiquem se alguma transição deve ocorrer. Este fluxo de controle representa o tratamento de duas atividades diferentes realizadas no meta-nível (delegação de serviço dependente de estado para um objeto de estado corrente e transição de estado), o que é feito por metaobjetos diferentes.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

MetaTransition

Classe *MetaTransition*	Colaboradores
• Possui todas as informações que definem uma transição. • Possui uma referência para o metaobjeto *MetaState* que representa o próximo estado a ser ativado. • Verifica se um evento provoca uma transição, podendo verificar também condições de guarda. • Ao verificar que uma transição deve ser efetuada, envia ao metaobjeto *MetaController* uma referência para o *MetaState* que representa o próximo estado.	No meta-nível: • *MetaState* • *MetaController*

O metaobjeto *MetaTransition* possui todas as informações necessárias que definem a função de transição que representa, ou seja, o estado corrente, o evento que ativa a transição e possíveis condições de guarda, e o próximo estado que deve ser ativado. O método *handleTransition()* recebe como parâmetro a mensagem que foi interceptada e materializada, verificando se a mensagem representa um evento significativo que provoca a transição (pela assinatura do método) podendo verificar também possíveis condições de guarda associadas ao objeto de contexto. Caso se identifique como sendo a transição que deve ser realizada, envia uma mensagem para o metaobjeto *MetaController* mudar o seu estado corrente, passando como parâmetro a referência para o *MetaState* que representa o novo estado. Para manter a consistência da máquina, apenas uma transição é ativada por vez.

Estrutura dinâmica

O diagrama de seqüência de eventos da Figura 2 mostra o cenário para o tratamento de um evento que é enviado para o objeto de contexto no nível base. O diagrama mostra as principais interações entre os metaobjetos para a realização dos aspectos de controle. A seqüência mostra um cenário em que um único evento representa uma solicitação de um serviço que é dependente de estado, e que portanto deve ser executado pelo objeto de estado do nível base, e ao mesmo tempo provoca uma transição de estado. As interações entre os objetos do meta-nível e do nível base são feitas utilizando-se o protocolo de metaobjetos(MOP).

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

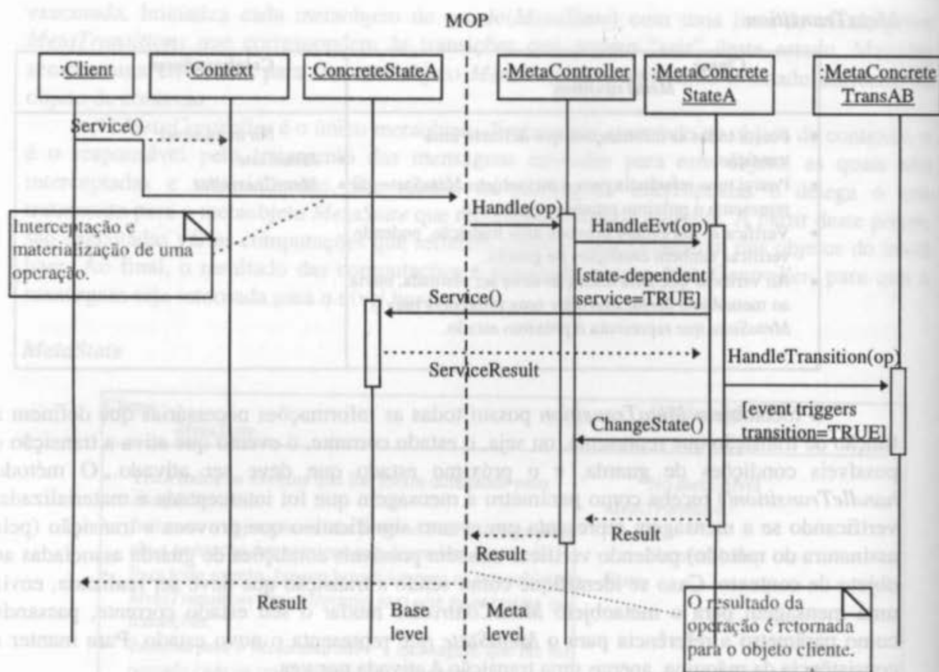

Figura 2: Diagrama de seqüência de eventos para a execução dos aspectos de controle realizados no meta-nível.

4.2 Conseqüências

A principal vantagem do uso do padrão *State* Reflexivo é a separação dos aspectos de controle da execução de *statecharts* dos aspectos funcionais implementados pelo objeto de contexto e pelos objetos de estado. A utilização da arquitetura reflexiva proporciona esta separação de uma forma transparente, tornando o projeto mais estruturado e fácil de entender, facilitando também a manutenção, extensão e reutilização. As hierarquias de classes do objeto de contexto e dos objetos de estado não possuem nenhum relacionamento, o que permite que sejam reutilizadas e estendidas separadamente.

A estrutura de controle definida no meta-nível para implementar os *statecharts* pode ser projetada de forma genérica, de modo que possa ser especializada com o mínimo de esforço para a implementação de um *statechart* específico para uma determinada classe de contexto. Isto permite que esta estrutura genérica seja totalmente reutilizada.

A principal desvantagem da aplicação do padrão *State Reflexivo* é o aumento de indireções para a execução dos métodos de um objeto, o que pode acarretar numa possível perda de eficiência. A medida do *overhead* causado pelas interceptações de mensagens pelo meta-nível depende da implementação do protocolo de metaobjetos utilizado.

4.3 Padrões Relacionados

Na literatura, existem alguns padrões recentes que discutem os problemas relacionados à implementação do padrão *State*. O trabalho de Dyson e Anderson [DA96] apresenta uma

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

linguagem de padrões de estado que classifica o padrão *State* em sete padrões relacionados, os quais refinam ou extendem este padrão. A linguagem de padrões também discute os aspectos de implementação do controle das transições de estados e da inicialização dos objetos de estado. Entretanto, estes padrões não separam a implementação dos aspectos de controle de transições dos aspectos funcionais.

O trabalho de J. Odrowski and P. Sogaard [OS96] define algumas variações do padrão *State* que resolvem problemas de implementação da dependência entre estados de objetos relacionados. Este trabalho mostra soluções para o problema de combinação do padrão *State* com outros padrões, entretanto, não discute o problema do controle das transições.

O trabalho de A. Ran [Ran95] apresenta uma família de padrões de projeto que também podem ser usados para resolver problemas de implementação de comportamento dependente de estado. Este padrão é apresentado na forma de uma árvore de decisão de projeto (DDT), que separa comportamentos dependente de estado em classes de Estado, e implementa o controle das transições e condições de guarda explicitamente usando métodos de transição e classes predicativas, respectivamente. O padrão *State* Reflexivo propõe a implementação das transições através dos metaobjetos *MetaTransition*, os quais encapsulam as informações sobre sua respectiva função de transição.

O artigo de G. Palfinger [Pal97] apresenta uma extensão do padrão *State*, definindo um objeto *State Mapper* que faz o mapeamento de eventos em ações. Este padrão usa uma lista de pares de eventos e ações, os quais podem ser adicionados/modificados/deletados em tempo de execução, provendo adaptação dinâmica para novos requerimentos. De uma forma similar, o padrão *State* Reflexivo também permite adaptação para novos requerimentos em tempo de execução usando o protocolo de metaobjetos para implementar mudanças da máquina de estado, tal como adição e deleção de estados e transições.

Existem outros trabalhos que fazem uso do conceito de reflexão computacional para a separação da parte de controle da parte funcional da aplicação. O trabalho de M. de Champlain [Cha96] apresenta um padrão que implementa máquinas de estado no meta-nível, implementando a parte de controle das transições e do estado corrente através de um metaobjeto. O objetivo deste padrão é semelhante ao objetivo do padrão *State* reflexivo, mas a estrutura é diferente, pois ele implementa todo o controle da máquina de estado em um único metaobjeto, enquanto o padrão *State* Reflexivo procura implementar separadamente cada elemento que compõe a máquina de estado, ou seja, os estados e as transições.

O trabalho de M. Coelho, C. Rubira e L. Buzato [CRB97] apresenta uma abordagem reflexiva para a implementação de *frameworks* para interfaces homem-computador. Este trabalho mostra uma uma reestruturação do *framework* ET++ [WGM89] em uma arquitetura reflexiva, implementando a parte de controle do padrão MVC no meta-nível. Com esta separação de controle no meta-nível diminuem-se as dependências entre os componentes modelo, visão e controle, definidos no nível base. O padrão *State* Reflexivo tem um objetivo semelhante que consiste em tornar os componentes da aplicação (objeto de contexto e objetos de estado) mais independentes e, consequentemente, mais fáceis de serem estendidos e reutilizados.

5. Variações do padrão State Reflexivo

A seguir, é apresentada uma das variações do padrão *State* Reflexivo que consiste no padrão *State Reflexivo* para o domínio de tolerância a falhas. No final deste tópico, é apresentado também um sistema de padrões formado pelo padrão *State Reflexivo*, e todas as suas variações, considerando tolerância a falhas de software, de hardware e de ambiente.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

5.1 Variação do padrão State Reflexivo para o domínio de Tolerância a Falhas

Um sistema tolerante a falhas é aquele projetado para prover os seus serviços confiavelmente apesar da ocorrência de falhas parciais durante a sua execução[LA90]. Existem basicamente três categorias de falhas que podem ocorrer em um sistema: falhas de hardware, que são falhas em componentes físicos do sistema; falhas de software, que são falhas nos componentes de software causadas por *bugs* nas fases de projeto, implementação ou manutenção durante o ciclo de desenvolvimento do software; e falhas de ambiente, que são falhas nas entidades do ambiente em que o sistema está inserido, e com as quais ele interage.

Todo mecanismo de tolerância a falhas é baseado na introdução e aplicação de redundância no sistema, para que este possa continuar a prover os serviços mesmo na presença de falhas, aumentando a confiablidade[1] e disponibilidade[2] do sistema [LA90]. Idealmente, os componentes redundantes devem ser introduzidos de uma forma estruturada e não-intrusiva, de forma a não aumentar a complexidade do sistema [Rub94].

Com base nas similaridades existentes no controle de redundância empregado pelos mecanismos de tolerância a falhas de hardware, software e de ambiente, foi definido um padrão genérico que faz o controle dos componentes tolerantes a falhas e dos componentes reduntantes, correspondendo a uma variação do padrão *State Reflexivo* para o domínio de tolerância a falhas. Na definição desta variação, utiliza-se a mesma estrutura do padrão *State Reflexivo*, mudando-se apenas as responsabilidades das classes. O objeto de contexto corresponde à classe do componente tolerante a falhas (*TFComponent*); a hierarquia de classes de estado representa os componentes redundantes; e o fluxo de controle do *statechart* representa a forma como os componentes reduntantes podem ser alternados ou compostos, de forma que o componente tolerante a falhas forneça serviços confiáveis (Figura 3). Os metaobjetos *MetaState* e *MetaTransition* são os responsáveis por fazer toda a computação adicional relacionada a implementação de uma determinada técnica de tolerância a falhas.

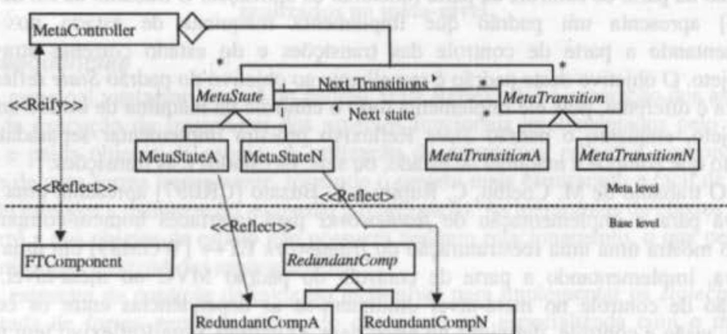

Figura 3. Diagrama de classes do padrão *State* Reflexivo para o domínio de Tolerância a Falhas.

5.2 Sistema de padrões para o desenvolvimento de software tolerante a falhas

O padrão apresentado acima corresponde a um padrão genérico, que pode ser customizado para padrões mais concretos que apresentam soluções para a implementação de tolerância a falhas de software, de hardware, e de ambiente. Para cada categoria de tolerância a

[1] Em Inglês: *reliability*
[2] Em Inglês: *availability*

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

falhas, pode-se ainda especificar padrões que implementam técnicas específicas, como por exemplo: em tolerância a falhas de software, pode-se utilizar as técnicas de N-Versões ou Blocos de Recuperação.

O conjunto destes padrões formam um sistema de padrões que ajudam na construção de software tolerante a falhas, considerando as três categorias de falhas: falhas de hardware, de software e de ambiente. Os padrões são fortemente relacionados, como mostra a árvore de relacionamento abaixo (Figura 4). Cada nível da árvore representa um nível de abstração dos padrões, os quais estão ligados por relacionamentos de refinamento, variação ou composição.

A árvore tem como base o padrão de arquitetura *Reflection* [BMRS+96] que estabelece a estrutura geral de todos os outros padrões. Este padrão combinado com o padrão de projeto *State*, deram origem ao padrão *State Reflexivo* e a todos os outros. A primeira variação do padrão *State Reflexivo* trata do problema do controle de redundância em sistemas tolerantes a falhas, como foi mostrado na seção 5.1, dando origem ao padrão "*State Reflexivo* para o domínio de Tolerância a Falhas". Este padrão é um padrão genérico que pode ser customizado para implementar tolerância a falhas de hardware, de software e de ambiente, dando origem a padrões mais concretos. Num último nível da árvore, tem-se os padrões que implementam técnicas específicas para cada tipo de tolerância a falhas, como por exemplo, o padrão que implementa a técnica de N-Versões para prover tolerância a falhas de software.

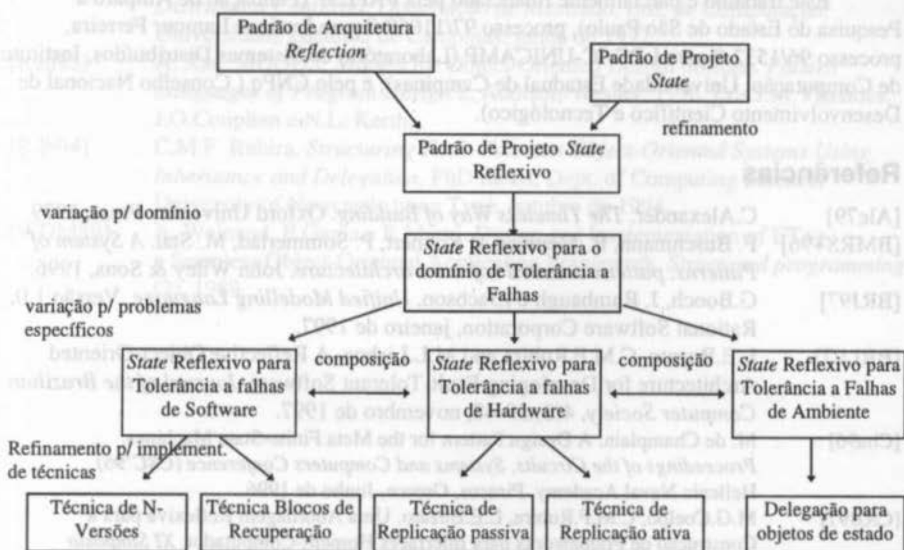

Figura 4: Árvore de relacionamento do sistema de padrões para o domínio de tolerância a falhas.

6. Conclusões

Este trabalho apresenta o resultado de um estudo detalhado da aplicação de padrões para auxiliar no processo de desenvolvimento de software orientado a objetos. Os problemas relacionados à aplicação do padrão *State* no desenvolvimento de software mais complexo levou à definição do padrão *State Reflexivo*, e de suas variações para o domínio de tolerância a falhas. Estes padrões agrupados formam um sistema de padrões para este domínio, baseados

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

em um padrão de arquitetura reflexiva. Este sistema de padrões está em fase de "amadurecimento", e o uso dos padrões no desenvolvimento de aplicações tolerantes a falhas pode apontar novos problemas, e conseqüentemente, novos padrões ainda podem surgir. Atualmente, os padrões estão sendo aplicados no desenvolvimento de um *framework* para o domíno de controladores de trens, e serão implementados mecanismos de tolerância a falhas de ambiente, de software e de hardware, mostrando-se como os diversos padrões podem ser combinados. Uma versão do controlador de trens foi desenvolvida em [Rub94], e no seu projeto, encontra-se um exemplo de uso da variação do padrão *State* para implementação de tolerância a falhas de ambiente. O trabalho de E. Quadros [Qua97] [QR97] apresenta uma primeira versão do projeto do *framework* para o domínio de controladores de trens, que também utiliza o padrão *State* no projeto dos componentes tolerantes a falhas adaptáveis. Estes dois trabalhos serviram como motivação para o estudo do padrão *State* e dos problemas associados à sua implementação. A implementação do framework usando os padrões propostos está sendo feita no Instituto de Computação da Unicamp, usando a linguagem de programação Java e o protocolo de metaobjetos Guaraná[OGB98].

Agradecimentos

Este trabalho é parcialmente financiado pela FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo), processo 97/11060-0 para Luciane Lamour Ferreira, processo 96/1532-9 para LDS-IC-UNICAMP (Laboratório de Sistemas Distribuídos, Instituto de Computação, Universidade Estadual de Campinas); e pelo CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico).

Referências

[Ale79] C.Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[BMRS+96] F. Buschmann, R. Meunier, H Rohnert, P. Sommerlad, M. Stal. *A System of Patterns: pattern-oriented software architecture*. John Wiley & Sons, 1996.

[BRJ97] G.Booch, J. Rambaugh e I.Jacbson. *Unified Modelling Language*. Versão 1.0, Rational Software Corporation, janeiro de 1997.

[BRL97] L.E.Buzato, C.M.F.Rubira and M.L.Lisboa. A Reflective Object-Oriented Architecture for Developing Fault-Tolerant Software. *Journal of the Brazilian Computer Society*, 4(2):39-48, novembro de 1997.

[Cha96] M. de Champlain. A Design Pattern for the Meta Finite-State Machines. *Proceedings of the Circuits, Systems and Computers Conference (CSC'96)*, Hellenic Naval Academy, Piraeus, Greece, Junho de 1996.

[CRB97] M.G.Coelho, C.M.F.Rubira, L.E.Buzato. Uma Abordagem Reflexiva para a Construção de Frameworks para Interfaces Homem-Computador. *XI Simpósio Brasileiro de Engenharia de Software (SBES'97)*, pag. 115-130, Fortaleza, CE, Outubro de 1997.

[DA96] P.Dyson and B. Anderson. State Patterns. *Pattern Languages of Program Design 3*, Addison-Wesley, 1997. Eds. R.Martin, D.Riehle, F.Buschmann.

[GHJV95] E.Gama, R. Helm, R Johnson e J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley Publishing, 1995.

[Har87] D.Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8: 231-274, North-Holland, 1987.

[LA90] A . Lee e T. Anderson. *Fault Tolerance: Principles and Practice*, Springer Verlag Wien New York, 1990.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

[Mae87] P.Maes. Concepts and Experiments in Computational Reflection. *Proc. OOPSLA'87*, ACM SIGPLAN Notices, 22(12):147-155, outubro de 1987.

[OB98] A.Oliva, L.E.Buzato. An Overview of MOLDS: A Meta-Object Library for Distributed Systems. Technical Report IC-98-15, Instituto de Computação, Universidade Estadual de Campinas, Abril 1998.

[OGB98] A.Oliva, I.C.Garcia, L.E.Buzato. The Reflexive Architecture of Guaraná. Technical Report IC-98-14, Instituto de Computação, Universidade Estadual de Campinas, abril 1998.

[OS96] J. Odrowski and P. Sogaard. Pattern Integration - Variations of State. *PLoP'96 Writer's Workshop.* (http://www.cs.wustl.edu/~schmidt/PLoP-96/Worshops.html)

[Pal97] G.Palfinger. State Action Mapper. *PLoP'97 Writer's Workshop* (http://st-www.cs.uiuc.edu/hanmer/PLoP-97/Workshops.html).

[QR97] E.M.Quadros e C.M.F. Rubira. Construção de um Framework para Sistemas Controladores de Trens Utilizando Padrões de Projeto e Metapadrões. *XI Simpósio Brasileiro de Engenharia de Software (SBES'97)*, Fortaleza, CE, Outubro 1997.

[Qua97] E.M. Quadros. *Uma Abordagem Orientada a Objetos para Programação Distribuída Confiável.* Dissertação de Mestrado, Inst. de Computação, Universidade Estadual de Campinas, junho de 1997.

[Ran95] A. Ran. MOODS: Models for Object-Oriented Design of State. *Pattern Languages of Program Design 2*, Addison-Wesley, 1996. Eds.J.M.Vlissides, J.O.Couplien e N.L. Kerth.

[Rub94] C.M.F. Rubira. *Structuring Fault-Tolerant Object-Oriented Systems Using Inheritance and Delegation*. PhD thesis, Dept. of Computing Science, University of Newcastle upon Tyne, outubro de 1994.

[WGM89] A. Weinand, E.Gama e R.Marty. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured programming*, (2), 1989.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor