Um *Framework* Orientado a Objetos para Controladores de Trens Tolerantes a Falhas

Luciane Lamour Ferreira

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Um *Framework* Orientado a Objetos para Controladores de Trens Tolerantes a Falhas

Luciane Lamour Ferreira[1]

Agosto de 1999

Banca examinadora:

- Profa. Dra. Cecília Mary Fischer Rubira (Orientadora)
  Instituto de Computação – UNICAMP
- Prof. Dr. Carlos José Pereira de Lucena
  Departamento de Ciência da Computação – Puc Rio
- Profa. Dra. Eliane Martins
  Instituto de Computação – UNICAMP
- Prof. Dr. Luiz Eduardo Buzato
  Instituto de Computação – UNICAMP

[1] Auxílios concedidos pela FAPESP, processo no. 97/11060-0 e pelo CNpQ, processo no. 131962/97-3

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Ferreira, Luciane Lamour

F413f Um framework orientado a objetos para controladores de trens tolerantes a falhas / Luciane Lamour Ferreira -- Campinas, [S.P. :s.n.], 1999.

Orientadora : Cecília Mary Fischer Rubira

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Framework (Programa de computador). 2. Engenharia de software. 3. Software - Desenvolvimento. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Instituto de Computação
Universidade Estadual de Campinas

Um *Framework* Orientado a Objetos para Controladores de Trens Tolerantes a Falhas

Este exemplar corresponde à redação final da dissertação de mestrado devidamente corrigida e defendida por Luciane Lamour Ferreira, e aprovada pela banca examinadora.

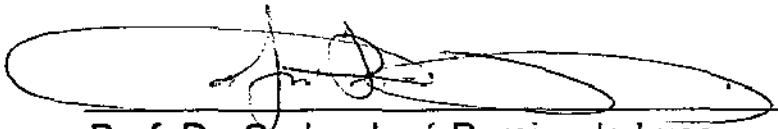
Campinas, 29 de setembro de 1999

Cecília Mary Fischer Rubira
Orientadora: Cecília Mary Fischer Rubira

Dissertação de mestrado apresentada ao Instituto de Computação da Universidade Estadual de Campinas – Unicamp, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 29 de setembro de 1999, pela Banca Examinadora composta pelos Professores Doutores:



Prof. Dr. Carlos José Pereira de Lucena
PUC - Rio

Eliane Martins

Profa. Dra. Eliane Martins
IC - UNICAMP

Cecília Mary Fischer Rubira

Profa. Dra. Cecília Mary Fischer Rubira
IC - UNICAMP

Agradecimentos

À toda a minha família, e em especial aos meus pais, Mário e Walkíria, pela confiança, pelo apoio e pelo sacrifício que sempre fizeram para que eu pudesse me dedicar aos estudos.

Ao meu marido, Rodrigo, pelo amor, carinho e compreensão, e por estar sempre ao meu lado nos momentos mais difíceis, me apoiando e incentivando incondicionalmente.

Aos amigos do LSD, pelo companheirismo e amizade demonstrados em todos os dias em que compartilhamos o mesmo espaço de trabalho.

À minha orientadora, Cecília Rubira, pela forma coerente com que conduziu este trabalho.

Aos professores e funcionários do Instituto de Computação – Unicamp, pelos serviços prestados ao programa de mestrado.

À Fundação de Amparo à Pesquisa do Estado de São Paulo – FAPESP e ao Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNpQ, pelo apoio financeiro.

Resumo

Este trabalho baseia-se nos conceitos de orientação a objetos, *frameworks*, estilos de arquitetura, padrões de projeto e metapadrões, para o projeto e implementação de um *framework* orientado a objetos para controladores de trens tolerantes a falhas e distribuídos. O principal objetivo é a obtenção de reutilização de software em larga escala, com reutilização tanto de código quanto de todo o projeto de software. No desenvolvimento do *framework*, nós utilizamos estilos de arquitetura para o projeto da sua parte fixa, e padrões de projeto e metapadrões para a documentação da sua parte adaptável. Nosso objetivo é avaliar as vantagens e desvantagens obtidas na aplicação destas técnicas na construção de *frameworks*. Este trabalho apresenta também propostas de novos padrões de projeto e estilos de arquitetura, que foram utilizados para resolver problemas do domínio do *framework*. A principal contribuição dos padrões e estilos é a utilização de reflexão computacional na implementação de tolerância a falhas, com o objetivo de obter estruturas de projeto mais flexíveis, o que é uma característica essencial para obtenção de *frameworks* realmente reutilizáveis.

Abstract

This work is based on the concepts of object-orientation, frameworks, architectural styles, design patterns and metapatterns to the design and implementation of an object-oriented framework for fault-tolerant train controlers. The main goal is to obtain large-scale reuse, reusing not only the code but also the whole software design. In the framework development, we have applied architectural styles in the design of its fixed parts, and design patterns and metapatterns in the design of its adaptable parts. Our goal is to evaluate the advantages and disadvantages of applying these tecniques in the framework construction. This work also presents new design patterns and architectural styles that have been used to solve problems in the framework domain. The main contribution of the patterns and styles is the use of computational reflection in the fault tolerance implementation in order to achieve more adaptable design structure, which is an essential feature of frameworks.

Conteúdo

Lista de figuras por artigo

Capítulo 1

Introdução

A crescente complexidade de sistemas de software modernos tem levado pesquisadores na área de engenharia de software a procurar soluções que visam manter a complexidade sob controle, e ao mesmo tempo melhorar o processo de desenvolvimento de software. A reutilização de software tem sido proposta como uma das abordagens mais promissoras para a solução de tais problemas, devido aos benefícios que ela traz como, por exemplo, aumento da qualidade de software e redução do custo de seu desenvolvimento. Um dos principais benefícios advindo da adoção do paradigma de orientação a objetos refere-se à sua capacidade potencial de aumentar o grau de reutilização de software [RBP+92]. No entanto, a reutilização efetiva de componentes de software não é uma tarefa trivial, pois os componentes de software devem ser adequadamente projetados para serem reutilizáveis [JF88]. Além disto, uma reutilização efetiva não se baseia apenas na reutilização de código, mas também na reutilização de todo o projeto de software. Vários conceitos têm sido propostos para a obtenção de um grau de reutilização razoável, como por exemplo, *frameworks* [JF88, Joh97a], padrões de projeto [GHJV95], padrões e estilos de arquitetura [SG96] e metapadrões [Pre95].

Um *framework* proporciona uma arquitetura de software "pré-fabricada" para um domínio de aplicações, proporcionando adaptabilidade apropriada para que várias aplicações com características específicas possam ser criadas através da sua reutilização. Podemos identificar duas fases principais no desenvolvimento de um *framework*: (i) o projeto de sua arquitetura, que compreende as principais classes e o fluxo de controle do domínio e (ii) o projeto dos pontos adaptáveis que serão instanciados para atender às características de aplicações específicas. Para se obter um software genérico que seja ao mesmo tempo fácil de ser

compreendido, reutilizado e também mantido (com modificações em seu próprio projeto), é necessário a utilização de técnicas de projeto adequadas durante o seu desenvolvimento.

O projeto da arquitetura de um *framework* é uma tarefa complexa, visto que um *framework* deve incluir a funcionalidade de uma família de aplicações, incluindo requisitos funcionais e não-funcionais. Os requisitos funcionais referem-se à funcionalidade básica de uma aplicação, enquanto os requisitos não-funcionais referem-se às propriedades que descrevem como uma aplicação deve satisfazer suas funcionalidades básicas, como por exemplo, tolerância a falhas, distribuição, etc. Estilos de arquitetura têm sido utilizados para a definição da arquitetura de software, oferecendo soluções para a estruturação geral de software e um conjunto de restrições que guiam todo o processo de seu desenvolvimento. Em relação ao desenvolvimento de *frameworks*, estilos de arquitetura podem ser utilizados na definição da sua parte fixa, definindo os principais componentes que implementam os requisitos funcionais e não-funcionais do domínio, o relacionamento entre eles, as principais restrições e o fluxo de controle geral.

A documentação dos pontos adaptáveis é uma etapa essencial para se obter uma efetiva reutilização do *framework*, visto que o usuário do *framework* deverá compreender as estruturas adaptáveis para que ele possa estendê-las e/ou configurá-las para a implementação de uma aplicação específica. Padrões de projeto e metapadrões têm sido propostos como uma forma de documentar os pontos adaptáveis de um *framework*. A reutilização das soluções propostas pelos padrões proporciona uma redução nos esforços de projeto do *framework*, e ao mesmo tempo melhora o entendimento do mesmo. Metapadrões constituem uma abordagem mais abstrata de padrões cujo principal objetivo é a documentação das estruturas flexíveis de um *framework*, identificando mais explicitamente quais são os pontos fixos e os pontos adaptáveis. Eles podem ser utilizados para documentar outros padrões de projeto em um meta-nível, constituindo portanto uma abordagem complementar à padrões de projeto na documentação dos pontos adaptáveis de um *framework*.

O principal objetivo deste trabalho é a aplicação destas técnicas de reutilização de software, através do desenvolvimento de um *framework* orientado a objetos para o domínio de controladores de trens tolerantes a falhas e distribuídos, com o intuito de comprovar a efetividade destas técnicas. No desenvolvimento do *framework*, nós utilizamos estilos de arquitetura para o projeto da parte fixa do *framework*, e padrões de projeto e metapadrões para a documentação dos

seus principais pontos adaptáveis. Durante a experiência prática de utilização de padrões de projeto e estilos de arquitetura no desenvolvimento deste software complexo, nós encontramos problemas que não eram satisfatoriamente resolvidos por padrões de projeto e estilos de arquitetura existentes. Neste contexto, nós apresentamos também propostas de refinamentos e variações de padrões existentes, e documentamos um novo estilo de arquitetura para o domínio de tolerância a falhas.

Em resumo, os principais objetivos deste trabalho são:

- Projeto e implementação de um *framework* orientado a objetos para o domínio de controladores de trens tolerantes a falhas e distribuídos. O principal objetivo deste projeto é o desenvolvimento de um estudo de caso que nos permita utilizar as principais técnicas relacionadas à reutilização de software em larga escala.
- Utilização prática de padrões de projeto e metapadrões na documentação dos pontos adaptáveis do *framework*, e a análise das vantagens e limitações destas abordagens quando utilizadas em conjunto.

As principais contribuições do nosso trabalho são:

- Documentação do padrão *Reflective State* [FR98a, FR98c], que é um refinamento do padrão de projeto *State* [GHJV95] utilizando o padrão de arquitetura *Reflection* (também conhecido como estilo *Meta-Level*) [BMRS+96]. O padrão proposto oferece uma estrutura mais flexível do que a estrutura do padrão *State* original, sendo portanto mais adequado na documentação de pontos adaptáveis de *frameworks*.
- Documentação de um sistema de padrões para o domínio de tolerância a falhas formado por um conjunto de variações do padrão *Reflective State*. O uso de reflexão computacional permite que os mecanismos de tolerância a falhas sejam executados de forma transparente para os objetos que implementam a funcionalidade básica da aplicação.
- Documentação do estilo de arquitetura *Idealized Fault-Tolerant Component* baseado no modelo de estruturação de sistemas tolerantes a falhas de mesmo nome, proposto por Lee e Anderson [LA90]. A documentação deste modelo como um estilo de arquitetura contribui para a reutilização desta solução no projeto de arquiteturas de software tolerantes a falhas. Este estilo também foi utilizado no projeto da arquitetura do *framework* para controladores de trens.

- Como produto final, nós implementamos um *framework* orientado a objetos para controladores de trens que inclui mecanismos para tolerância a falhas e distribuição. Este *framework* foi implementado na linguagem Java, utilizando o Guaraná para a implementação dos componentes reflexivos. O Guaraná [Oli98] é uma arquitetura de software reflexiva que permite um alto grau de reutilização de código de meta-nível, tendo sido implementado na linguagem Java. A implementação do *framework* para controladores de trens foi uma experiência prática importante que contribuiu para a avaliação da efetividade das técnicas de reutilização de software em larga escala.
- A implementação do padrão *Reflective State* na linguagem Java, utilizando a arquitetura do Guaraná, também representa um *framework* para implementação de máquinas de estados no meta-nível, o qual pode ser reutilizado independentemente do *framework* para controladores de trens.

Organização da dissertação:

Os capítulos desta dissertação são compostos de artigos escritos em inglês que foram submetidos e publicados em conferências e revistas internacionais de grande relevância na área de engenharia de software. Os artigos descrevem o trabalho realizado durante o mestrado, e os principais resultados obtidos. Cada capítulo é formado por uma introdução em português, pelo(s) artigo(s) correspondente(s), e um resumo do capítulo também em português.

O **capítulo 2** apresenta os padrões de projeto que foram documentados como propostas novas de padrões e que foram utilizados no projeto dos pontos adaptáveis do *framework*. Os padrões apresentam soluções para problemas encontrados durante o projeto do *framework*, principalmente relacionados à tolerância a falhas, e que não eram satisfatoriamente resolvidos por padrões existentes. Estes padrões são descritos em dois artigos. O artigo "*The Reflective State Pattern*" apresenta um refinamento do padrão *State* [GHJV95] utilizando o estilo de arquitetura *Meta-Level*. Este padrão apresenta uma solução para o problema relacionado à implementação das transições de estados do padrão *State*, propondo a implementação de uma máquina de estados no meta-nível, de forma que esta seja executada de uma forma transparente para os objetos da aplicação. Esta solução gera uma estrutura de projeto mais flexível do que a solução original do padrão *State*, sendo portanto mais adequada para a documentação dos pontos adaptáveis de um *framework*. O artigo "*Reflective Design Patterns to Implement Fault*

Tolerance" apresenta um sistema de padrões formado por um conjunto de variações do padrão *Reflective State* para o domínio de tolerância a falhas. Este sistema de padrões discute como a mesma estrutura do padrão *Reflective State* pode ser utilizada com semânticas diferentes para a implementação das técnicas de tolerância a falhas de software, de hardware e de ambiente.

O **capítulo 3** contém o artigo "*Architectural Styles and Patterns for Developing Dependable Frameworks*", que discute dois estilos de arquitetura que foram utilizados no projeto da arquitetura do *framework*: (i) o estilo "*Idealized Fault-Tolerant Component*", que foi documentado como um novo estilo de arquitetura e (ii) o estilo "*Meta-Level*". Nós mostramos como estes dois estilos auxiliam na definição da arquitetura de sistemas tolerantes a falhas, com o principal objetivo de manter a complexidade sob controle. Nós também introduzimos o projeto do *framework* para controladores de trens e definimos sua arquitetura utilizando estes estilos.

O **capítulo 4** descreve o projeto detalhado do *framework* utilizando estilos de arquitetura para a definição da sua parte fixa e padrões de projeto e metapadrões para a documentação da sua parte adaptável. Para isto, utilizamos os estilos e padrões descritos nos capítulos anteriores, e também outros padrões existentes na literatura. O projeto é descrito no artigo "*The Design and Implementation of a Dependable and Distributed Framework for Train Controllers*", que apresenta também as principais conclusões obtidas neste experimento prático.

O **capítulo 5** apresenta as conclusões do nosso trabalho, apresentando as principais contribuições e os possíveis trabalhos futuros.

Capítulo 2

Padrões de Projeto

Padrões de projeto constituem boas soluções de projeto para problemas recorrentes dentro de um contexto particular. A documentação de padrões facilita o entendimento destas soluções e sua efetiva reutilização em uma grande variedade de domínios de software. A documentação destes padrões em catálogos [GHJV95, BMRS+96] e a crescente popularização de conferências especializadas em padrões, como por exemplo PLoP e EuroPLoP, permitem que engenheiros de software utilizem uma referência confiável no desenvolvimento de seus projetos, reutilizando a experiência de outros desenvolvedores e soluções boas e previamente testadas. As conferências especializadas em padrões promovem a documentação de novos padrões e o amadurecimento de padrões existentes.

Durante o projeto do *framework* para controladores de trens, nós utilizamos vários padrões de projeto existentes que oferecem soluções flexíveis, entre eles, o padrão *State* [GHJV95]. Este padrão oferece uma solução para a implementação de serviços que são dependentes de estado, através da definição de uma hierarquia de classes de estado paralela à hierarquia de classes do objeto da aplicação. O objeto da aplicação mantém a referência para o objeto de estado corrente, e delega para ele a execução dos seus métodos. Para mudar o seu estado, o objeto da aplicação muda a referência para o objeto de estado corrente. Entretanto, a implementação do padrão *State* gera alguns problemas que não são bem discutidos na sua documentação, como por exemplo, o problema de onde definir e executar as transições de estados que representam a execução de uma máquina de estados. As possíveis soluções discutidas na documentação do padrão *State* tornam as classes de estado e do objeto da aplicação fortemente acopladas, o que dificulta suas futuras extensões. Para resolver este problema, nós propomos o padrão *Reflective State*, que utiliza uma arquitetura reflexiva para a implementação

de uma máquina de estados no meta-nível, de forma que esta seja implementada separadamente e executada de forma transparente para os objetos da aplicação.

Este capítulo é composto de dois artigos que documentam esta nossa solução em duas instâncias. O primeiro artigo, "*The Reflective State Pattern*", apresenta o refinamento do padrão *State* em uma arquitetura reflexiva, apresentando uma solução genérica que pode ser utilizada no mesmo contexto em que o padrão *State* é utilizado. O artigo foi publicado na "*5th Pattern Language of Programs Conference (PLoP'98)*" realizada em Monticello, Illinois, EUA, em agosto de 1998.

O segundo artigo denominado "*Reflective Design Patterns to Implement Fault Tolerance*" apresenta um sistema de padrões reflexivos para o domínio de tolerância a falhas. Estes padrões são variações do padrão *Reflective State* que apresentam soluções para a implementação das técnicas de tolerância a falhas de hardware, de software e de ambiente. Na implementação dos pontos adaptáveis para tolerância a falhas do *framework*, nós utilizamos a variação do padrão *Reflective State* para tolerância a falhas de ambiente. Este artigo foi publicado em "*proceedings of the Workshop on Reflective Programming in C++ and Java: Workshop # 13 of OOPSLA'98*", realizado em Vancouver, Canadá, em outubro de 1998.

The Reflective State Pattern[1]

Luciane Lamour Ferreira Cecília M. F. Rubira
Institute of Computing - State University of Campinas
P.O. Box 6176, Campinas, SP 13083-970
e-mail: {972311, cmrubira}@dcc.unicamp.br

Abstract

This paper presents the Reflective State pattern that is a refinement of the State design pattern [GHJV95] based on the Reflection architectural pattern [BMRS+96]. This pattern proposes a solution for some design decisions that have to be taken in order to implement the State pattern, such as the creation and the control of State objects and the execution of state transitions. When the object has a complex dynamic behavior, its implementation can also become very complex. The Reflective State pattern implements the control aspects in the meta level, separating them from the functional aspects that are implemented by the Context object and the State objects located at the base level. This pattern provides a solution that is easier to understand, extend and reuse than the State pattern.

1 Introduction

The State design pattern[GHJV95] is a well known pattern that has been used in various applications[JZ91] [Rub94]. Its purpose is to allow an object to change its behavior when its internal state changes, implementing state-dependent services by means of State classes and using the delegation mechanism. The implementation guidelines of this pattern discusses some design decisions that should be taken in order to implement the states and to control the transitions. Various patterns have discussed the implementation aspects of the State pattern, and have proposed refinements [DA96], variations [OS96] and extensions [Pal97]. However, when a class has a complex behavior, the implementation of the control aspects of the State pattern can also become complex.

The Reflection architectural pattern [BMRS+96] provides a mechanism for dynamically changing structure and behavior of software. This pattern separates an application in two levels: a base level and a meta level. The base level defines the application's logic where objects implement the functionalities as defined on its functional requirements. The meta level consists of meta-objects that encapsulate and represent information about the base-level objects. The meta-objects can perform management actions that dynamically interfere with the current computations of the corresponding base-level objects. The relationship among the base-level objects and the meta-level objects is specified by means of a meta-object protocol (MOP). Generally speaking, a meta-object protocol establishes the following interactions [Lis98]: (1) attachment of base-level and meta-level objects, that can be static or dynamic, and on a one-to-one or many-to-one meta-objects to base-level objects basis; (2) reification, which means materialization of information otherwise hidden from the program, such as incoming and outcoming messages, arguments, data and other structural information; (3) execution, which consists of meta-level computation that interferes in the base-level behavior transparently through the interception and reification mechanisms; (4) modification, which is the capability of the meta-objects of changing behavior and structural base-level aspects. The Reflection architectural pattern has been previously applied for achieving separation of concerns in various domains, such as fault tolerance [BRL97] [Lis98] and distribution [OB98][Buz94].

In this work we present the Reflective State pattern which uses the Reflection architectural pattern to separate the State pattern in two levels, the meta level and the base level, implementing the control of states and transitions by means of meta-objects. The Reflective State is a generic pattern that intends to solve the same problems of the State pattern in the same context. However, this pattern also implements complex dynamic behavior of an object transparently, separating its control aspects from the functional aspects.

2 The Reflective State Pattern

Intent

To separate the control aspects related to states and their transitions from the functional aspects of the State pattern. These control aspects are implemented in the meta level by means of the meta-objects which represent the state machine's elements (FSM elements).

Motivation

Consider the same motivation example of the State pattern[GHJV95]: a class TCPConnection that represents a network connection. For simplicity, we restrict the TCPConnection to having two basic states: established and closed. Depending on its current state, it can respond differently to the client's requests. For example, the implementation of an *open()* request depends on whether a connection is in its closed state or established state. Moreover, the TCPConnection object can change its current state when an event occurs or when a condition is satisfied. In this example, the same request *open()* also represents an event that causes a transition to the established state. There is also a guard-condition associated with this transition, meaning that the transition will only be triggered if the TCPConnection object has received an acknowledge. A transition can also have an action associated with it, that is performed in the moment the transition is triggered. In our example, when a TCPConnection changes to the established or closed state, it can display a message. Therefore, a request can represent a service that has a state-specific implementation and an event that causes a transition to the next state. This dynamic behavior is specified by the state diagram of Figure 1.

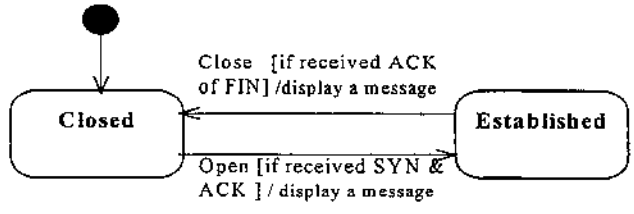


Figure 1: State diagram of the class TCPConnection using the UML notation.

Context

The behavior of an object depends on its internal state, so the implementation of its services can be different for each possible state. Furthermore, an object can have a complex dynamic behavior specified by a state diagram or a statechart[Har87]. The state diagram is composed by a triple (states, events/guard-conditions, state transitions). The state transition function depends on the current state and the input event and/or a guard-condition. The statechart extends the state diagram with the notion of hierarchy, concurrence and communication.

There are several contexts where the pattern can be applied, for instance, in reactive systems that receive events (outside stimuli) and respond to them by changing their state and,

consequently their behavior. Other examples of use can be in the context of distributed systems, control systems, graphical user's interface systems, fault-tolerant systems, etc. These systems can have classes with a complex dynamic behavior specified by a complex and large statechart. Ideally, the design and implementation of the corresponding state machine using object-oriented approach should be made in a structured manner, representing the states and their transitions as explicitly as possible, to reduce the complexity of the system.

Problem

We can use the State pattern to localize state-specific behavior in State subclasses, which implement the state-dependent services. The Context object delegates the execution of its services to the current State object. However, the implementation of the State pattern deals with design decisions related to the control aspects of the state machine. These decisions are summarized in the following questions:

(1) Where should the definition and initialization of the possible State objects be located?

(2) How and where should the input events and guard-conditions be verified?

(3) How and where should the execution of state transitions be implemented?

When the object has complex behavior, implementation resulting from these decisions can become complex as well. According to the implementation guidelines of the State pattern, the control aspects can be located either in the Context object or in the State objects. In the first approach, the Context object is responsible for creating and initializing the State objects, maintaining the reference to the current State object, and performing the transition to the next state. Moreover, it is also responsible for delegating the state-dependent service execution to the current State object. In the second approach, the State objects have the knowledge of their possible next states, and have the responsibilities for handling the events or conditions that causes transitions. Both approaches have some disadvantages:

- Centralizing the control aspects in the Context object can make its implementation very complex since its functional aspects are implemented together with the control aspects. Moreover, it makes the Context class highly coupled with the State classes, which makes it difficult to reuse and extend them.
- Decentralizing the responsibilities of the transition logic, by allowing State subclasses themselves specify their successor states, can make them highly coupled, introducing

implementation dependencies between State subclasses. It also prevents the State class hierarchy from being extended easily.

The following forces are related with these implementation problems:

- Ideally, the implementation of the control aspects of the State pattern should be separated from the functional aspects implemented by the Context object and the State objects. These control aspects should be implemented transparently, ideally in a non-intrusive manner, so that they do not complicate the design.
- The Context and State class hierarchies should be loosely coupled, to facilitate their reutilization and extension.
- The State subclasses should also be independent, with no implementation dependencies between them, so that the addition of a new subclass does not affect other subclasses.

Solution

To solve the problems related with the implementation aspects of the State pattern we propose the use of the Reflection architectural pattern [BMRS+96]. The State pattern can be separated in two levels: the finite state machine (FSM) level and the application level, which correspond to the meta and base level of Reflective architectures, respectively. The FSM-level classes are responsible for implementing the control aspects of the finite state machine, separating them from the functional aspects that are implemented by the Context class and the State classes at the application level, such as in the State pattern. In the FSM level, the elements of the state diagram (states and transitions) are represented by the FSMState and the FSMTransition class hierarchies. The state machine's controller is represented by the FSMController class. The interception and materialization mechanisms provided by the meta-objects protocol make the execution of the control aspects transparent, oblivious to the application-level objects. Figure 2 shows the class diagram of the Reflective State pattern, where the main contribution is above the line, at the FSM level.

Structure

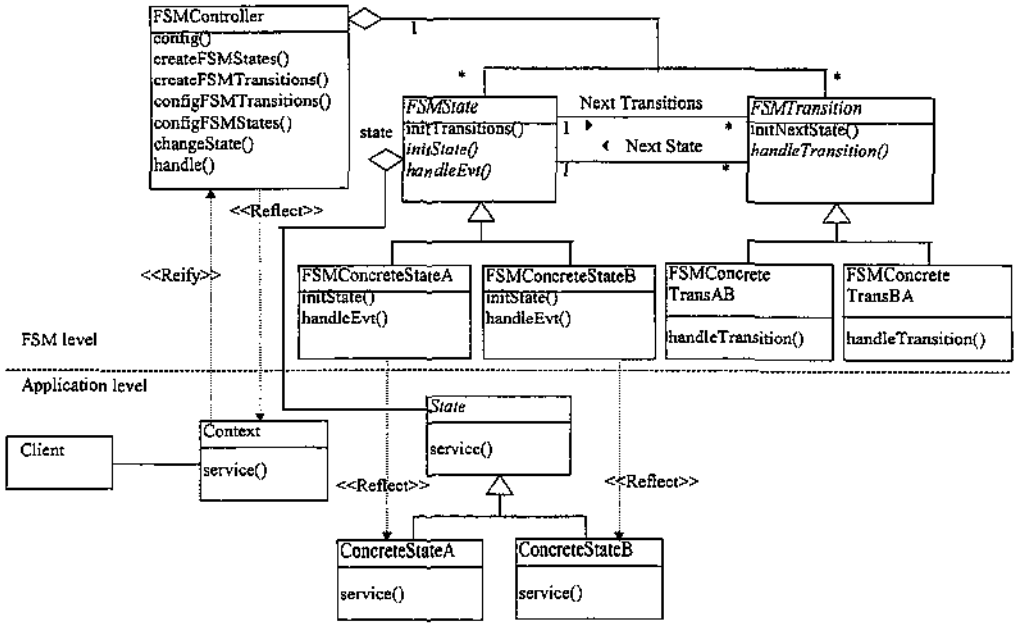


Figure 2: Class diagram of the Reflective State pattern using the UML notation.

To illustrate our solution, we can design the TCPConnection example using the Reflective State pattern structure. Figure 3 shows the object diagram for an instance of the TCPConnection class, with its respective State objects and meta-objects which implement the TCPConnection's state machine of Figure 1.

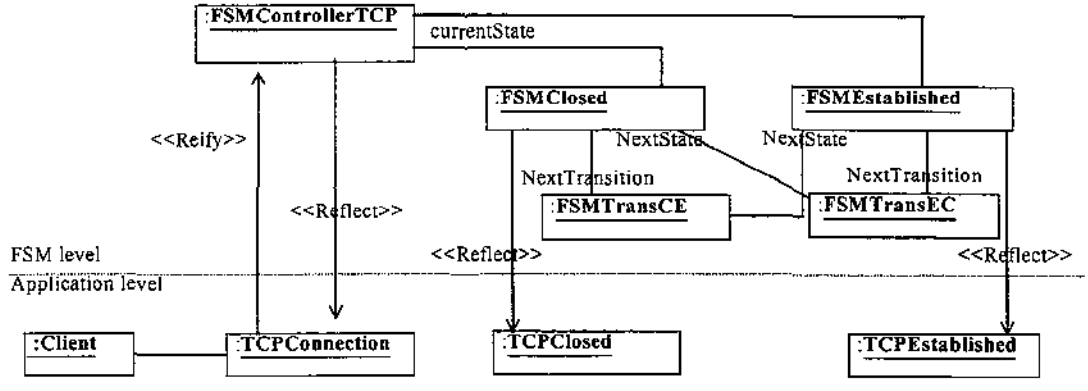


Figure 3: An object diagram for a TCPConnection instance, applying the Reflective State pattern.

The states of the TCPConnection are represented by the TCPEstablished object and the TCPClosed object at the application level, which implement the state-dependent services. The FSMEstablished and FSMClosed meta-objects are responsible for initializing their corresponding State objects, and controlling the execution of the state-dependent service. The FSMTransition meta-objects represent the transitions of the statechart and they are: FSMTransEC (established-to-closed transition) and FSMTransCE (closed-to-established

transition). Each FSMTransition has information about the transition function (the event, the guard-conditions and the exit action) that should be verified before a transition is triggered. The FSMControllerTCP meta-object maintains a reference to the current FSMState meta-object, and changes it when a FSMTransition signals that a transition has occurred. The FSMControllerTCP is responsible for intercepting and materializing all client's service requests targeted to the TCPConnection object.

Participants

The responsibilities of the pattern classes are presented using the CRC Cards notation.

FSMController

Class FSMController	**Collaborators**
• Configures the FSM level, instantiating and initializing the concrete FSMState and FSMTransition subclasses, according to the state diagram specification. • Intercepts all messages sent to the Context object. • Maintains the reference to the FSMState metaobject that represents the current state, and delegates to it the handling of the intercepted messages. • Performs the state transition, changing the reference to a new current FSMState metaobject, that is passed by a FSMTransition object.	In the FSM level: • FSMState • FSMTransition In the application level: • Context

FSMState

Class FSMState	**Collaborators**
• Defines an interface for handling an event that represents a state-dependent service. • Defines an interface for initializing the State object at the application level. • Defines a method that initializes itself with a list of FSMTransition references that represent the transitions that can exit from this state.	In the FSM level: • FSMTransition In the application level: • State

FSMConcreteState subclasses

Class FSMConcreteState	**Collaborators**
• Handles all events delegated to it by the FSMController. • Creates and initializes the corresponding State object at the application level, and delegates to it the execution of the state-dependent services. • Broadcasts each event to the FSMTransition metaobjects so that they can verify if the event causes a transition. • Receives the result of the service execution from the state object at the application level, and can also handle the result, if necessary. • Returns the result of the service execution to the FSMController.	In the FSM level • FSMConcreteTrans subclasses In the application level • ConcreteState subclasses

FSMTransition

Class FSMTransition	**Collaborators**
• Defines an interface to handle transitions. • Defines a method that initializes itself with a reference to a FSMState metaobject that represents the next state to be activated when the transition is triggered.	In the FSM level • FSMState • FSMController

FSMConcreteTrans subclasses

Class FSMConcreteTrans	**Collaborators**
• Has all information that defines a transition function, i.e., the current state, the event/guard-condition and the next state. • Verifies if an event causes a transition and/or if a guard-condition is satisfied. • If the transition is triggered, it requests the FSMcontroller metaobject to change its current state, passing to it the reference to the next FSMState.	In the FSM level • FSMConcreteState subclasses • FSMController

Context class

Class Context	**Collaborators**
• Defines the service interface of interest to clients, as defined in its functional requirements.	In the application level: • Others application classes

State class

Class State	**Collaborators**
• Defines an interface for encapsulating the behavior associated with a particular state of Context (as defined in the State pattern).	In the application level • Context

ConcreteState subclasses

Class ConcreteState	**Collaborators**
• Each subclass implements a behavior associated with a state of the Context(as defined in the State pattern).	In the application level • Context

Collaborations

The meta-objects represent a direct mapping of the state diagram elements. The configuration of the FSM level consists of: instantiation of each concrete subclass of the FSMState class and FSMTransition class; initialization of the FSMState meta-objects with their corresponding FSMTransitions meta-objects; initialization of each FSMTransition meta-object with its corresponding next FSMState meta-object. The FSMController meta-object is responsible for implementing all these configurations according to the state diagram's specification of a Context class.

The meta-objects are responsible for controlling the execution of state-dependent services and the state transitions (Figure 4). The interactions between the FSM meta-objects and the application-level objects are performed by means of a meta-object protocol (MOP). The MOP's kernel should implement the interception and materialization mechanism so that the meta-objects can perform the extra computation related to the execution of the state machine. There are several different MOPs implementations, thus we do not show a specific MOP's interaction; instead we make the assumption that a MOP's implementation performs all communications

between the base and meta levels transparently, and materializes the operation with its basic information. The materialized operation can be represented as an object, which should be passed as a parameter of the *handle()* method. In Figure 4, the dotted line represents a MOP's implementation. After the operation is materialized, the MOP's kernel delivers it to the FSMController meta-object which initializes the handling of the materialized operation.

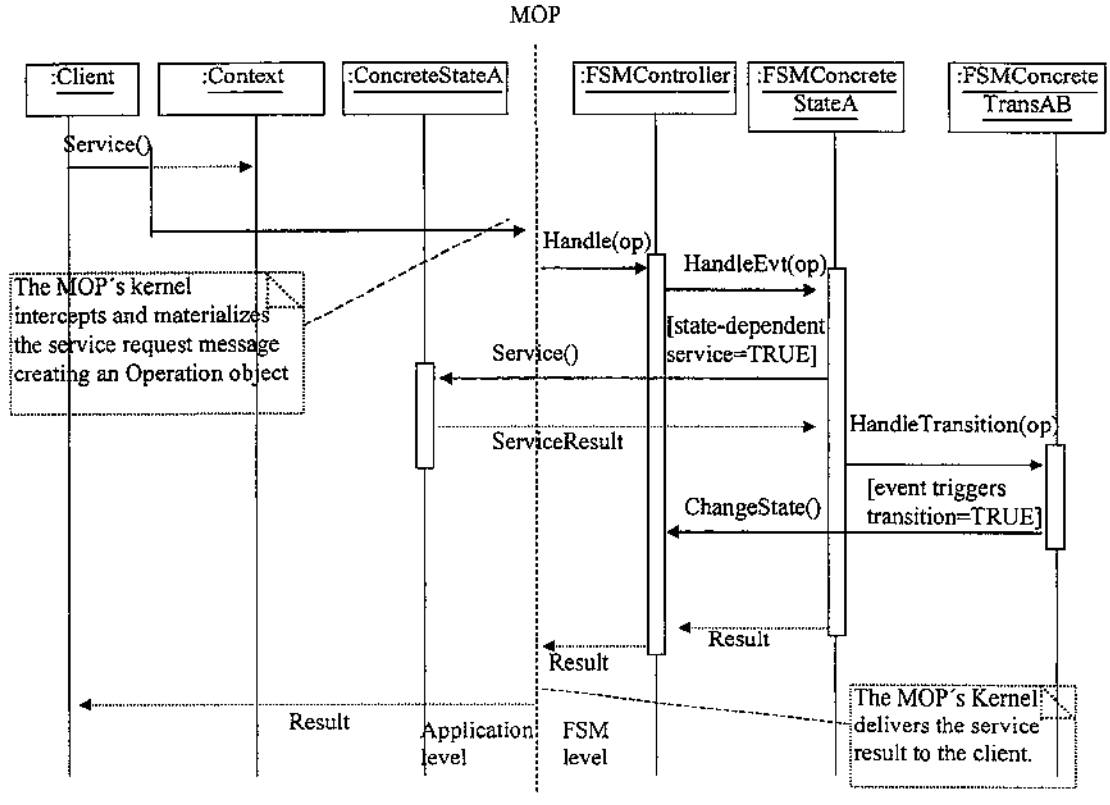


Figure 4: Interaction diagram for the Reflective State Pattern

The FSMController meta-object intercepts the service request targeted to the Context object and delegates its handling to the current FSMConcreteState meta-object. The current FSMConcreteState meta-object verifies if the request corresponds to a state-dependent service. If so, the current FSMConcreteState meta-object delegates the service execution to its corresponding ConcreteState object at the application level. It also delegates the event's handling to the FSMConcreteTrans meta-object so that it can decide whether the message corresponds to an event that causes a transition or not. A FSMConcreteState can have a list of FSMTransitions, and it can delegate the handling of the transition sequentially or concurrently. If a FSMConcreteTrans meta-object verifies that its transition should be triggered, it requests the

FSMController meta-object to change its current State, passing to it the reference to the next FSMConcreteState meta-object. After the service request has been handled, the service result is first returned to the FSMController meta-object, and then to the MOP's Kernel which delivers the result to the client.

Consequences

The Reflective State pattern localizes state-specific behaviors and partitions behavior for different states, as in the case of the State pattern. The state objects make implementation of the state-dependent services more explicit, and consequently, the design becomes more structured and easier to understand, maintain and reuse.

The Reflective State provides a solution for implementing the control aspects of the State pattern, separating them from the functional aspects implemented by the Context object and State objects. This characteristic is provided by the Reflection architectural pattern. This solution makes the implementation of the dynamic behavior of a class (that might be specified by a complex state diagram) more explicit, also making the design more structured, keeping the complexity of the system under control.

The State and Context class hierarchies are independent and they can be designed to be highly coherent and loosely coupled, facilitating the adaptability of the system to the changes of requirements, its reuse and extension.

The Reflective State pattern has some limitations related to the use of the reflective architecture. In general, a reflective architecture increases the number of indirections in the execution of a method, causing an impact in the system's performance.

Implementation

The Reflective State pattern proposes some criteria for the implementation decisions that have to be taken in order to implement the State pattern. It also adds other implementation decisions, mainly related to the use of reflective programming.

1. *Where should the definition and initialization of the possible State objects be located?* The State objects can be defined and instantiated by their respective FSMState meta-objects. The State objects' creation can be implemented using the Factory Method pattern. The FSMState

hierarchy at the FSM level corresponds to the State hierarchy at the application level. The abstract FSMState class defines an interface for creating State objects, the abstract method *initState()*, that should be overridden in the FSMState subclasses, so that the concrete State objects can be created. This solution makes the extension of the State hierarchy easier: if a concrete State class is added, a FSMState class should also be added so that it can instantiate this State object. The FSMController can be reconfigured dynamically, adding a new FSMState object at runtime. A new FSMTransition meta-object that refers to the new FSMState meta-object can also be added in the same manner. If dynamic reconfiguration is desired, the interface of the FSMController should define methods for adding FSMStates and FSMTransitions. Some MOP can also implement the reconfiguration mechanism in a more transparent manner.

Another possible solution is to implement only one generic concrete FSMState class when the *handleEvt()* method does not need to be overridden to implement specific behavior related to a concrete State class. Then, instead of using subclassing to create the possible State objects by means of the Factory Method *initState()*, one can parameterize the FSMState instance with the concrete State object that the FSMState should refer to. In languages that treat classes as first-class objects, like Smalltalk and Java, the FSMState instances can be initialized with the respective concrete State class object, that can be the creator of its instances, and acts like prototypes. Using parameterized FSMState meta-objects instead of FSMState subclasses reduces the number of classes of the FSM level, although the number of instances is the same. It also makes reconfiguration process easier, since new states can be added by instantiating new FSMState objects parameterized with the new concrete State class, instead of using subclassing to create the new State object.

As discussed in the State pattern implementation section, the State objects can be implemented as the Singleton pattern, so that many FSMState meta-objects of different meta state machines can share the same application-level State objects.

2. *How and where should the input events and guard-conditions defined in the state machine be verified*? The Reflective State pattern establishes a well-defined criterion for this issue. The events of the state machine correspond to the method calls or field accesses targeted to the Context object, that are materialized and presented to the meta-objects. The FSMTransition

meta-object is responsible for verifying the input event (the name of the method, the arguments, the result type) and the guard-conditions associated with the transition. The guard-conditions are boolean expressions that associate incoming arguments, attribute's values or methods of the Context object. If the transition is verified, the FSMTransition meta-object should request the FSMController meta-object to change its current state, passing a reference to next FSMState as a parameter.

3. *Where should the configuration of the FSM level be performed?* The FSMController class interface defines some methods which perform the configuration of the state machine at meta level. The *CreateFSMStates()* and *CreateFSMTransitions()* methods should instantiate each meta-object according to the states and transitions of the Context class' statechart. The *ConfigFSMStates()* method should configure each FSMState meta-object with its corresponding next FSMtransition meta-objects and the *ConfigFSMTransitions()* method should configure each FSMTransition meta-object with a reference to its next FSMState meta-object. In order to obtain reusability, the FSMController class can be implemented as an Abstract Factory pattern. The FSMController class can define abstract methods to create and configure the meta-objects that implement a specific state machine. The concrete subclasses of the FSMController should override these methods creating and configuring the FSMState and the FSMTransition meta-objects according to a statechart specification for a specific Context class. This solution also makes the extension of the Context class easier: if a Context subclass is defined and the statechart is extended, a new FSMController subclass should also be defined, and it can redefine the configuration methods so that new FSMState and FSMTransition subclasses can be instantiated according to the new statechart specification.

These three implementation guidelines have assumed basic MOP's characteristics. Thus, it is very important to choose a MOP that implements these basic characteristics and gives a full support to the main reflection mechanisms. The MOP should implement the interception and reification mechanisms, so that the meta-objects can inspect the service request targeted to the Context object at the application level, and also perform the extra computation related to the execution of the state machine. Ideally, the MOP should also provide some base classes that can

be derived by the meta-objects, so that they can implement meta level behavior. For instance, the classes of the meta level can be derived from a Meta-object base class, which defines the default behavior of the meta-objects, such as the method *handle()*, called by the MOP's kernel. Other useful base classes may be a class to represent an operation (a service request materialized with its essential information), or a class to represent the result of an operation that has been performed. The result objects can also be presented to the meta-objects so that they can inspect, modify or replace them.

The pattern can also be implemented using a more restrictive MOP, but it requires harder programming work and it can also impose some restrictions. For instance, one can implement the FSM level classes and the application level classes separately, and can make the connection between the two levels explicitly through delegation mechanisms, instead of using the interception mechanism provided by the MOP. This solution makes the control aspects of the FSM execution less transparent and more intrusive for the application classes, but it can be a possible solution, maybe better than to implement the FSM control within the application classes.

Known Uses

The implementation of state machines has been widely discussed in the development of reactive systems. These systems tend to be very large and complex, and the implementation of a complex state machine is not a trivial task. It has motivated the study of the state machine implementation based on an object-oriented approach [SM92] implementing the states and transitions more explicitly. This approach has also been discussed in many related design patterns [DA96, OS96, Ran95, Pal97]. The use of the computational reflection to implement the state machine has been discussed in [deC96]. This work also implements the control of the transitions at the meta level, and defines the State objects and the Context object at the base level.

We are applying the Reflective State pattern in the development of a framework for the environmental fault-tolerant train controller's domain. An environmental fault-tolerant train controller system implements some components that represent environmental entities that may be faulty, such as sensors and switches. These components, in the solution domain, should reflect the normal and abnormal behavior phase of the environmental entities. The work of Rubira

[Rub94] proposes a solution for the design of environmental fault-tolerant components using the State pattern, implementing the normal and abnormal behavior phases by means of State objects. In the framework's implementation, we should also consider some requirements such as extensibility to specific application requirements. The extensibility requirement has been the motivation for our study of the State pattern design decisions. Using the Reflective State pattern, the fault-tolerant component hierarchy (related to the Context hierarchy in the general structure) and the State hierarchy become more independent, and consequently, easier to extend and reuse. Also, the execution of the transitions and state-dependent services become more explicit, making the design easier to understand, which are essential features of a framework.

We are using the Reflective State pattern to implement the environmental fault-tolerant components of the framework; in the future, we intend to implement other fault-tolerance requirements, such as software and hardware fault tolerance. We have defined a System of Patterns for the fault-tolerant domain [FR98] which has the Reflective State pattern as the most generic pattern. Other patterns of the System derive from the Reflective State pattern structure, adding fault-tolerance semantics.

To implement the framework, we are using the MOP of Guaraná [OGB98], which emphasizes flexibility, reconfigurability, security and meta-level code reuse. We are using a free Java-based implementation of the Guaraná reflective architecture that is currently available, and has been developed in the Institute of Computing of the State University of Campinas.

Related patterns

In the literature, there are some recent patterns that have discussed the implementation problems related to the State pattern. The work of Dyson and Anderson [DA96] presents a state pattern language that classifies the State pattern into seven related patterns that refine and extend this pattern. The pattern language also discusses the implementation aspects of the state transition control and the initialization of the State objects. However, these patterns do not separate the implementation of the transition control aspects from the functional aspects.

The work of Odrowski and Sogaard [OS96] defines some variations of the State pattern that solve implementation problems related to objects' state and the dependency between states of related objects. This work shows solutions for the problem of combining the State pattern with other patterns, however it does not discuss the problem of transition control.

Alexander Ran's work [Ran95] presents a family of design patterns that can be used to cope with the implementation of complex, state-dependent representation and behavior. This pattern family is presented in the form of a design decision tree (DDT), that separates state-behavior in State classes, and implements the control of transitions and guard-conditions explicitly, using transition methods and predicative classes, respectively. The Reflective State pattern proposes the separation of these control aspects by means of the FSMTransition meta-objects that encapsulate the information about the transition functions.

The paper by Günther Palfinger [Pal97] presents an extension of the State pattern, defining a State Mapper object that maps events to actions, using a list of event/action pairs. The list can be added/modified/deleted at runtime, providing easy adaptation to new requirements dynamically. In a similar manner, the Reflective State pattern can also be adapted dynamically using the meta-object protocol to implement changes of the state machine, such as addition of new states and transitions.

Sample Code

We exemplify the implementation of the TCPConnection class using Guaraná's MOP [OGB98]. It defines a base class, called Meta-object, that encapsulates the meta-level behavior, providing all interface and essential implementation for a meta-object so that it can handle operations, results, etc. The meta-object protocol of Guaraná establishes that an object can be directly associated with either zero or one meta-object, and this association is dynamic. The kernel of Guaraná implements a method *reconfigure()* that associates an object with a meta-object, and can also replace an old meta-object with a new one, allowing dynamic meta reconfiguration. The MOP of Guaraná also defines a more specialized kind of meta-object, the Composer, which groups meta-objects that are commonly used together and delegates the operation's handling to them. The Composers allow many meta-objects to be indirectly associated with an object. The meta-objects that are directly and indirectly associated with an object form its meta configuration. A more specialized kind of Composer is the SequentialComposer that delegates the operation's handling sequentially.

The following example presents a partial Java code for the TCPConnection example, using the Guaraná's MOP [OGB98]. The meta-object classes that implement the state machine

are derived from the following base classes of Guaraná: Meta-object, Composer and SequentialComposer.

FSM level classes

FSMController class: The FSMController class is derived from the Composer class of Guaraná, since the FSMController groups the meta-objects that implement the meta state machine, and delegates the operation's handling to them. In fact, the FSMController meta-object delegates only to the FSMState objects which are also Composers (SequentialComposers) that delegate to the FSMTransitions meta-objects.

```
import BR.unicamp.Guarana.*;

public abstract class FSMController extends Composer{
   protected Meta-object[] fsmStatesArray;
   protected FSMState currentFSMState;

   protected abstract void createFSMStates();
   protected abstract void createFSMTransitions();
   protected abstract void configFSMStates();
   protected abstract void configFSMTransitions();

   public final void config(){
         createFSMTransitions();
         createFSMStates();
         configFSMTransitions();
         configFSMStates();
   }
   public void changeState(FSMState nextState){
         currentFSMState = nextState;
   }
   public Result handle(Operation operation, Object object){
         if (operation.isConstructorInvocation())
         // returning "null"  means that the meta-objects do not handle constructor invocation
            return null;
         return currentFSMState.handle(operation,object);
   }
}
```

FSMControllerTCP class:

```
import BR.unicamp.Guarana.*;

public class FSMControllerTCP extends FSMController{
   protected  FSMEstablished fsmEstablished;
   protected  FSMClosed fsmClosed;
   protected  FSMTransEC fsmTransEC;
   protected  FSMTransCE fsmTransCE;

   protected void  createFSMStates(){
         fsmEstablished = new FSMEstablished();
         fsmClosed = new FSMClosed();
         currentFSMState = fsmClosed;       //initializes with a default state.
   }

   protected void createFSMTransitions(){
         fsmTransCE = new FSMTransCE(this);
         fsmTransEC = new FSMTransEC(this);
   }

   protected void configFSMTransitions(){
         //Configures the FSMTransitions with its next FSMStates
         fsmTransCE.initProxState(fsmEstablished);
         fsmTransEC.initProxState(fsmClosed);
   }

   protected void configFSMStates(){
         //Configures the FSMState meta-objects with the array of next FSMTransitions meta-object
         fsmEstablished.initTransitions(new FSMTransition[]{fsmTransEC});
         fsmClosed.initTransitions(new FSMTransition[]{fsmTransCE});

         //initiliazing the array of FSMStates that the FSMController delegates to.
         fsmStatesArray = new FSMState[]{fsmClosed,fsmEstablished};
   }
}
```

FSMState abstract class:

```
import BR.unicamp.Guarana.*;
```

```
import java.lang.reflect.*;

public abstract class FSMState extends SequentialComposer{
   protected State stateObject;

   protected abstract void  initState(Object object);

   public void initTransitions(FSMTransition[] arrayNextTransitions){
         //calls the method in the SequentialComposer base class.
         super.setMeta-objectsArray(arrayNextTransitions);
   }
}
```

FSMEstablished concrete class

```
import BR.unicamp.Guarana.*;
public class FSMEstablished extends FSMState{
   public void initState(){
         stateObject = new TCPEstablished();
   }

   public Result handle(Operation operation,Object object){
         //Verifies if an operation is a state dependent service.
         String name = operation.getMethod().getName();
         Class[] parameters = operation.getMethod().getParameterTypes();
         //it can modify the parameter array if the state method defines another parameter, as a
         //TCPConnection reference.
         //....
         Result res = null;
         if (stateObject == null) initState();          //it's initialized only if it's necessary.
         if (operation.isMethodInvocation()){
            Object resultObj;
            try {
                  //returns a public method of the class.
                  Method methodEx = stateObject.getClass().getMethod(name,parameters);
                  Object[]arguments = operation.getArguments();
                  resultObj = methodEx.invoke(stateObject,arguments);
                  if (resultObj == null){
                     res = Result.returnVoid(operation);
                  }
                  else {
```

```
                    res = Result.returnObject(resultObj,operation);
                }
            }
            catch (IllegalAccessException e1){
                //do some exception handling
            }
            catch (NoSuchMethodException e2){}
            catch (InvocationTargetException e3){}
        }
        //Delegates the operation's handling to the FSMTransition meta-objects sequentially,
        // calling the handle() method of the SequentialComposer
        super.handle(operation,object);

        //can do some handling with the result, unless it is returned
        //.....
        return res;
    }
}
```

FSMClosed concrete class

```
import BR.unicamp.Guarana.*;

public class FSMClosed extends FSMState{
    public void initState(Object object){
        stateObject = new TCPClosed();
    }

    public Result handle(Operation operation,Object object){
        //Verify if an operation is a state dependent service, like the FSMEstablished class does.
        // It can inspect the result and does some handling.
     }
}
```

FSMTransition abstract class

```
import BR.unicamp.Guarana.*;
import java.lang.reflect.Method;
public abstract class FSMTransition extends Meta-object{
    protected FSMController fsmController;
    protected FSMState nextState;
```

```
    public FSMTransition(FSMController fsmController){
        this.fsmController = fsmController;
    }

    public void initProxState(FSMState nextState){
        this.nextState = nextState;
    }
}
```

FSMTransEC concrete class

```
import BR.unicamp.Guarana.*;
import java.lang.reflect.Method;

public class FSMTransEC  extends FSMTransition{
    public FSMTransEC(FSMController fsmController){
        super(fsmController);
    }

public Result handle(Operation operation,Object object){
        //define the transition function.
        String eventName = "close";
        protected int paramNum = 0;
        if (operation.isMethodInvocation()){
                Method opMethod = operation.getMethod();
                if ((eventName.equals(opMethod.getName())) &&
                  (opMethod.getParameterTypes().length == paramNum)){
                //the event is correct. It can also test some guard-conditions using the "object" //parameter
                //....
                fsmController.changeState(nextState);
                //if the event and guard-conditions are verified
                }
        }
        return null;
    }
}
```

FSMTransCE concrete class

```
import BR.unicamp.Guarana.*;
```

```
import java.lang.reflect.Method;

public class FSMTransCE  extends FSMTransition{
   public FSMTransCE(FSMController fsmController){
         super(fsmController);
   }

    public Result handle(Operation operation,Object object){
         //defines the transition function
         //verifies the transition testing the event name and arguments of the operation, and the
         //guard-conditions.
         //...
   }
}
```

Application-level classes

The TCPConnection class and its respective State classes implement only their functional requirements, without any information about the execution control of the state machine.

TCPConnection class: The state-dependent methods do not have any implementation. Optionally, they can present some default behavior that can be executed if a TCPConnection object has not been associated with a FSMControllerTCP meta-object.

```
public class TCPConnection{
   public TCPConnection(){}

   public void open(){
//some default behavior;
   }

   public void close(){}

   //other methods and attributes
}
```

TCPState class

```
public abstract class TCPState{
   //If there are some state attributes, defines them here
   public abstract close(TCPConnection);
```

```
    public abstract open(TCPConnection);
}
```

TCPEstablished class

```
public class TCPEstablished extends TCPState{
    public close(TCPConnection tcpCon){
        //it closes the connection
    }
    public open(TCPConnection tcpCon){
        //it does nothing, because the Connection is already open.
    }
}
```

TCPClosed class: The implementation is similar to the Established class.

TCPApplication class: This class represents the application class which implements the *main()* method. First, the *main()* method creates a FSMControllerTCP meta-object and calls the method *config()* that configures the FSMControllerTCP. Then, it creates a TCPConnection object and calls the Guaraná *reconfigure()* method (which implements the Guaraná's Kernel). The *reconfigure()* method receives three parameters: (1) a reference to the object to be reconfigured, in this case, the TCPConnection object; (2) a reference to an oldMeta-object, if the object has been already configured with another one, and in this case it is null; (3) a reference to a newMeta-object, which the object is being reconfigured with, in this case, the FSMControllerTPC meta-object.

```
public class TCPApplication{
    public static void main(String[] argv){
        FSMControllerTCP fsmControllerTCP = new FSMControllerTCP();
        fsmControllerTCP.config();
        TCPConnection aTCPConnection = new TCPConnection();
        Guarana.reconfigure( aTCPConnection, null, fsmControllerTCP);
    }
}
```

3 Acknowledgments

We would like to thank our shepherd Dr. Michel de Champlain for valuable comments and suggestions for improvement of the pattern.

This work is partially supported by FAPESP (*Fundação de Amparo à Pesquisa do Estado de São Paulo*), grant 97/11060-0 for Luciane Lamour Ferreira, grant 96/1532-9 for LSD-IC-UNICAMP (*Laboratório de Sistemas Distribuídos, Instituto de Computação, Universidade Estadual de Campinas*); and by CNPq (*Conselho Nacional de Desenvolvimento Científico e Tecnológico*), grant 131962/97-3.

4 References

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

[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, November, 1997.

[Buz94] L.E.Buzato. *Management of Object-Oriented Action-Based Distributed Programs*. Ph.D. Thesis, University of Newcastle upon Tyne, Department of Computer Science, December 1994.

[deC96] 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, June 1996.

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

[FR98] L.L.Ferreira and C.M.F.Rubira. Integration of Fault Tolerance Techniques: a System of Pattern to Cope with Hardware, Software and Environmental Fault Tolerance. *Digest of FastAbstracts: FTCS'28 (the 28th Annual International Symposium on Fault-Tolerant Computing),* June 23-25, 1998, Munich, Germany, pp. 25-26.

[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.

[JZ91] R.E.Johnson and J. Zweig. Delegation in C++. *Journal of Object-Oriented Programming*, 4(11):22-35, November 1991.

[Lis98] M.L.B.Lisboa. A New Trend on the Development of Fault-Tolerant Applications: Software Meta-Level Architectures. *Proceedings of the 1998 IFIP - International Workshop on Dependable Computing and its Applications*, Johannesburg, South Africa, January, 1998.

[OB98] A.Oliva, L.E.Buzato. An Overview of MOLDS: A Meta-Object Library for Distributed Systems. *Technical Report IC-98-15, Institute of Computing, State University of Campinas*, April 1998.

[OGB98] A. Oliva, I.C.Garcia, and L.E.Buzato. The reflexive architecture of Guaraná. *Technical Report IC-98-14, Institute of Computing, State University of Campinas*, April 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).

[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*. Ph.D. Thesis, Dept. of Computing Science, University of Newcastle upon Tyne, October 1994.

[SM92] S.Shlaer and S.J.Mellor. *Object Lifecycles: Modeling the World in States*. Prentice-Hall, New Jersey, 1992.

Reflective Design Patterns to Implement Fault Tolerance

Luciane Lamour Ferreira Cecília Mary Fischer Rubira
Institute of Computing - IC
State University of Campinas – UNICAMP
P.O. Box 6176, Campinas, SP 13083-970 Brazil
+55 019 788 5847
{972311,cmrubira}@dcc.unicamp.br

Abstract

This paper discusses an object-oriented approach based on design patterns and computational reflection concepts to implement non-functional requirements of complex systems. First, we present the Reflective State pattern that is a refinement of the State design pattern based on the Reflection architectural pattern. The main goal is to separate the control aspects of the state-machine implementation from the application's logic. Then, we present some variations of this pattern for the fault-tolerance domain. The set of these variations originates a system of reflective design patterns that helps the development of well-structured fault-tolerant systems.

Key words: Computational Reflection, Reflective Architecture, Design Patterns, Fault Tolerance.

1 Introduction

Modern object-oriented systems normally include various non-functional requirements that can increase the system's complexity. The development of such systems requires the use of appropriate techniques in order to control this additional complexity and to make the software more structured and easier to understand, maintain and reuse. In this paper we present an object-oriented approach based on design patterns and computational reflection concepts to implement non-functional requirements of complex systems. More specifically, we are considering control aspects of system whose objects present complex dynamic behavior, as reactive systems, fault-tolerant systems, distributed systems, etc. Two design patterns are discussed: the State design pattern [7] and the Reflection architectural pattern [1]. The State design pattern presents a

solution to implement state-dependent behavior of a Context object by means of state objects. It allows the Context object to change its behavior dynamically using the delegation mechanism. The Reflection architectural pattern defines a software architecture that separates an application into two parts: the base level, which implements the functional requirements, i.e., the application's logic, and the meta-level, which implements the control aspects.

We present the Reflective State pattern[6] that is a refinement of the State design pattern based on the Reflection architectural pattern. The Reflective State pattern applies the Reflection architectural pattern to implement a finite state machine in the meta-level, by means of metaobjects that represent state and transitions, and use the interception and materialization mechanisms for implementing the control aspects in a transparent manner. The Reflective State pattern is a generic and domain-independent pattern that can have variations to specific domains. We show the variation of this pattern to the fault tolerance domain, and present a system of patterns that helps the development of fault-tolerant systems, considering hardware, software and environmental fault tolerance. This system of patterns is being used in the development of a framework for the environmental fault-tolerant train controller's domain[11], and has been implemented using the Java programming language and the metaobject protocol (MOP) Guaraná [10]. The Guaraná's MOP has many features that allow a system to achieve high degree of flexibility, reconfigurability, security and meta-level code reuse, which are essential features for frameworks.

This work is organized as follows. The Section 2 gives an overview of the Reflective State pattern, discussing its four main elements: intents, context, problem and the proposed solution. The Section 3 presents the system of pattern for the fault tolerance domain, introducing the Software Redundancy pattern, which is the most general pattern for this domain, and its variations for the hardware, software and environmental fault tolerance.

2 The Reflective State Pattern

Intents

To separate the control aspects related to states and their transitions from the functional aspects of the State pattern. These control aspects are implemented in the meta-level by means of the metaobjects which represent the state machine's elements.

Context

The behavior of an object depends on its internal state, so the implementation of its services can be different for each possible state. Furthermore, an object can have a complex dynamic behavior specified by a state diagram or a statechart [8], which is composed by a triple (states, events/guard-conditions, state transitions).

There are several contexts where the pattern can be applied, for instance, in reactive systems[4] that receive events (outside stimuli) and respond to them by changing their state and, consequently their behavior. Other examples of use can be in the context of distributed systems[2], control systems[11], fault-tolerant systems[3] [11], etc. These systems can have classes with a complex dynamic behavior specified by a complex and large statechart.

Problem

The state machine's implementation for complex systems is not a trivial task. The control is normally implemented together with the functional aspects of the application, which complicate the design and make it difficult to understand, maintain and reuse. There are three main questions that should be considered in the state machine implementations: (1) Where should the definition and initialization of the possible State objects be located?; (2) How and where should the input events and guard-conditions be verified?; (3) How and where should the execution of state transitions be implemented?

When the object has a complex behavior, the implementation of these issues can also become very complex. Ideally, the implementation of the control aspects of state machine should be separated from the functional aspects implemented by the Context object and the State objects. Furthermore, these classes should be loosely coupled to facilitate their reutilization and extension.

Solution

We propose the use of the Reflection architectural pattern [1] to separate the State pattern structure in two levels, the meta-level and the base level. The meta-level classes are responsible for implementing the control aspects of the state machine, separating them from the functional aspects that are implemented by the Context class and the State classes at the base level. In the meta-level, the elements of the state diagram (states and transitions) are represented by the

MetaState and the MetaTransition class hierarchies. The state machine's controller is represented by the MetaController class. The interception and materialization mechanisms provided by the metaobjects protocol make the execution of the control aspects transparent, oblivious to the base-level objects. Figure 1 shows the class diagram of the Reflective State pattern. The classes' responsibilities are showed below.

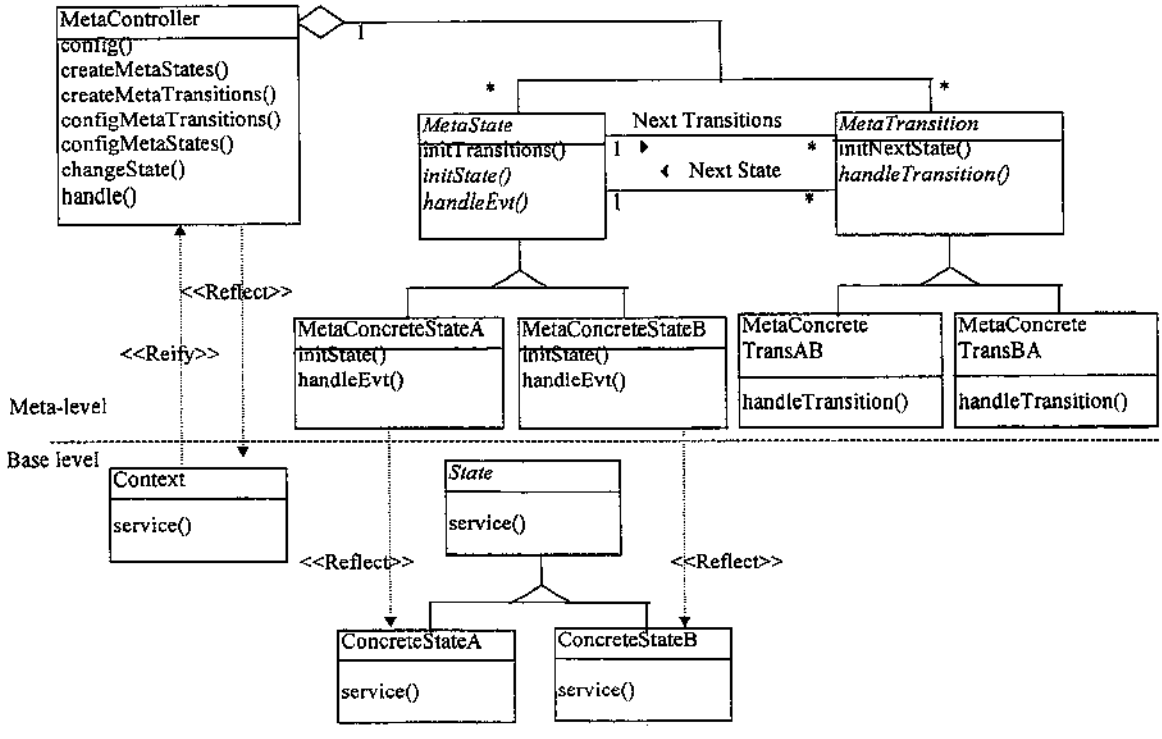


Figure 1: Class diagram of the Reflective State pattern using the UML notation.

MetaState: this class is responsible for creating and initializing the State objects at the base level. The MetaState metaobject receives a materialized service request and inspect it, verifying whether it is a state-dependent service. If so, it delegates the state-dependent service execution to its respective State object at base level. Also, the MetaState class has reference for a list of MetaTransitions that represent the transitions that exit from that state. The MetaState metaobject broadcasts the handling of the incoming event to its MetaTransition metaobjects so that they can verify if a transition should be triggered.

MetaTransition: this class represents the transitions of the state machine specified for the Context class. Each MetaTransition subclass has information about a transition function and is

responsible for verifying the incoming events and guard-conditions, and can also perform actions associated with the transition. The MetaTransition class keeps a reference to the next MetaState that can be reached by the transition. This reference is passed to the MetaController metaobject so that it can change the current state of the state machine.

MetaController: this class is responsible for handling the intercepted service requests targeted to the Context object at the base level. The MetaController metaobject is the primary metaobject associated to a Context object and is responsible for delegating the handling of the materialized operation to the current MetaState metaobject. In fact, the MetaController class represents the execution controller of the meta state-machine, and the materialized operations represent the incoming events for it. This class is also responsible for creating and configuring all metaobjects (metaconfiguration) that implements a specific statechart specification for a Context object.

3 A System of Patterns for the Fault Tolerance Domain

Techniques for achieving fault tolerance depend upon the effective deployment and utilization of redundancy[9]. The incorporation of redundancy in a software system requires a structured and disciplined approach, otherwise it may increase the complexity of the system and consequently it may decrease, rather than increase, the system's robustness. Ideally, one should consider the integration of hardware, software and environmental fault tolerance to cope with the various kinds of faults that can appear in a software system. Hardware fault tolerance[9] applies object replication to enhance the system availability/reliability in the presence of hardware faults; software fault tolerance[9] applies software redundancy by means of diversity of design to tolerate software faults that can occur at the design, programming or maintaining phases of the software development cycle, and environmental fault tolerance[11] copes with faults that can occur in real world entities in the problem domain and applies redundancy to represent the different abnormal behavior phases that the correspondent objects in the solution domain can present.

We propose a general pattern for the fault tolerance domain, which provides a uniform solution to the incorporation of redundancy in an object-oriented fault-tolerant system. This pattern, called the Software Redundancy pattern [5], defines a common structure that can be applied to the three kinds of fault tolerance to implement software redundancy. The Software

Redundancy pattern (Figure 2) presents the same structure of the State pattern, but with a different semantic. The base-level classes represent the fault-tolerant component and the redundant components. The former defines the fault-tolerant services and the later defines the mechanism to implement the redundancy. For instance, to cope with software faults (bugs) the RedundantComp subclasses implement different versions of the services provided by the FTComponent class. The meta-level classes implement the mechanisms correspondent to each fault tolerance technique. For instance, in the software fault tolerance, the MetaTransitions subclasses are responsible for implementing either the Acceptance Test of the Recovery-Block technique or the Voter of the N-Version technique. They should analyze the results and decide whether a service has been executed successfully or not.

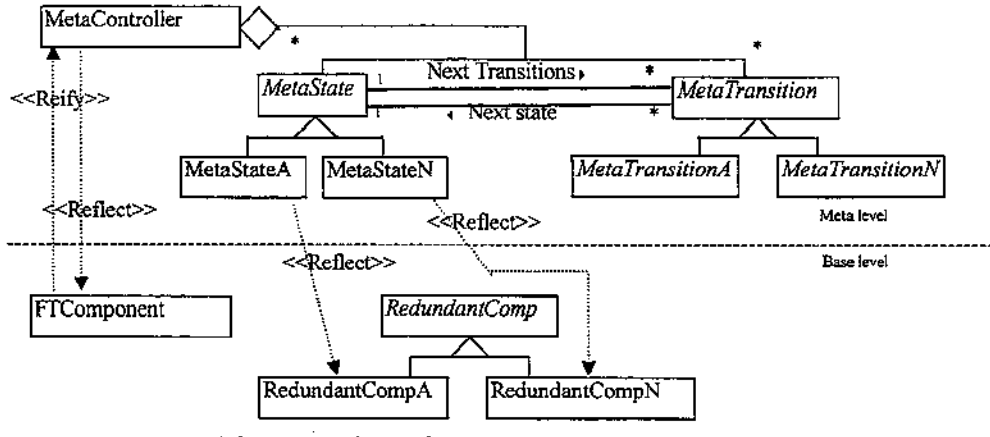


Figure 2: The Software Redundancy pattern [5]

This pattern is an abstract pattern that should be customized to implement environmental, software and hardware fault tolerance techniques, generating a system of patterns for the fault tolerance domain. These patterns are closely related to each other, as shown by the Patterns-Relationship Tree (Figure 3). Each tree's level represents a pattern's abstraction level, and the patterns are connected by relationship of refinement, variation or combination.

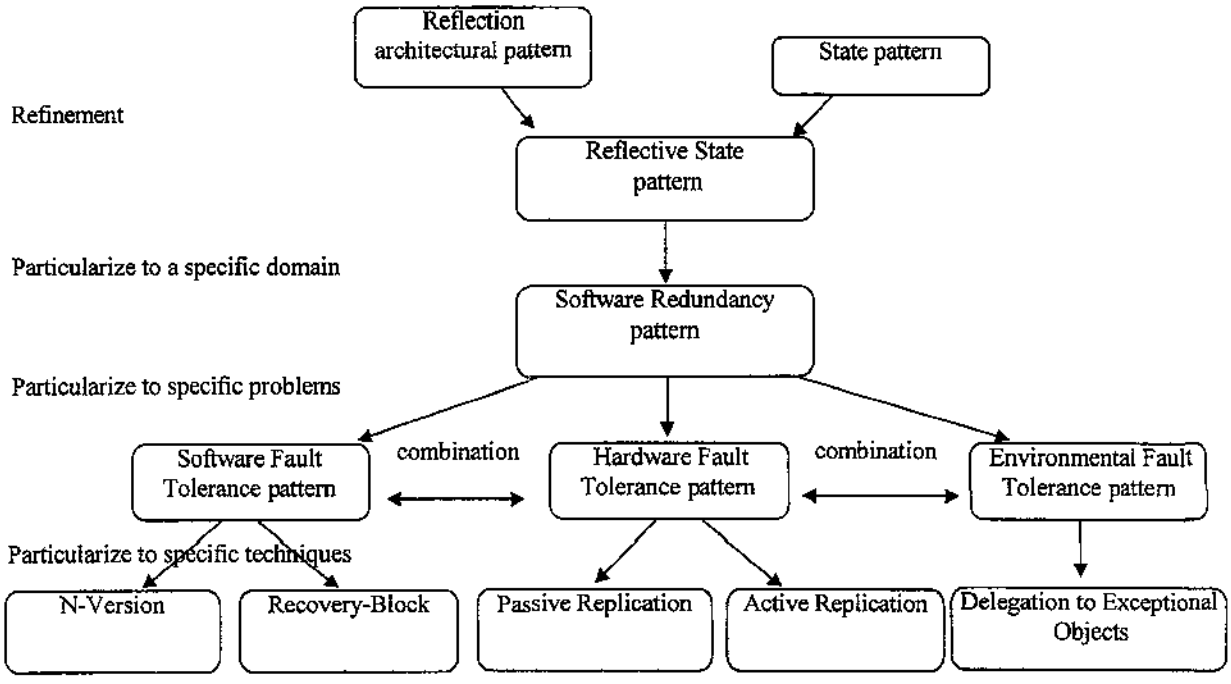


Figure 3: Patterns-relationship tree for the fault tolerance domain.

In the Environmental Fault Tolerance pattern, redundant components correspond to exceptional objects that encapsulate different service implementations, which represent the normal and abnormal behavior phases of these components [11]. A state transition occurs when an exception signals that the component has changed from the normal to the abnormal behavior phase. To handle a state dependent service, the current MetaState metaobject should delegate its execution to the state object at the base level. The current MetaState metaobject should also broadcast the event handling to the MetaTransitions metaobjects, so that they can verify if the event causes a state transition.

In the Software Fault Tolerance pattern, redundant components correspond to the different versions of the fault-tolerant component services. These versions are encapsulated by objects at the base level. A MetaState metaobject has a reference to a version object at base level and delegates to it the execution of the services. The result of the service execution is returned to the MetaState metaobject. Then the MetaState metaobject delegates this result for the MetaTransitions which handle them. For example, a MetaTransition metaobject can implement either the Acceptance Test of the Recovery-Block technique or the Voter of the N-Version technique.

In the Hardware Fault Tolerance pattern, the redundancy is provided by object replication. For instance, if a primary copy fails, a secondary copy can be executed to provide the same service. The redundant copies can be located in different computers in a distributed system, and the MetaState metaobjects are responsible for implementing the transparency of locality. The MetaState has a reference to the remote object, and should initialize it with the current state of the system and control the execution of the services through the network. The MetaTransitions are responsible for handling the run-time exceptions generated by a faulty copy, and activating a secondary copy.

These concrete design patterns can be combined to deal with hardware, software and environmental faults at the same time. A possible sequence for applying the patterns is the following. First, one can apply the environmental fault tolerance pattern to cope with environmental faults. Then, the software fault tolerance pattern can be applied to implement the n-versions of the state objects. Finally, the hardware fault tolerance pattern can be applied to implement the replication of the redundant components. Other combinations of these patterns are also possible to enhance the system reliability/availability, and they will depend on the system's requirements.

4 Acknowledgments

This work is partially supported by FAPESP (grant 97/11060-0) and CNPq (grant 131962/97-3) for Luciane Lamour Ferreira; and by FAPESP (grant 96/1532-9) for LSD-IC-UNICAMP (*Laboratório de Sistemas Distribuídos, Instituto de Computação, UNICAMP).*

5 References

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

[2] Buzato, L.E. *Management of Object-Oriented Action-Based Distributed Programs.* Ph.D. Thesis, University of Newcastle upon Tyne, Department of Computer Science, December 1994.

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

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

[5] Ferreira, L.L., and Rubira, C.M.F. Integration of Fault Tolerance Techniques: a System of Pattern to Cope with Hardware, Software and Environmental Fault Tolerance. *Digest of FastAbstracts: FTCS'28 (the 28th Annual International Symposium on Fault-Tolerant Computing),* June 23-25, 1998, Munich, Germany, pp. 25-26.

[6] Ferreira, L.L., and Rubira, C.M.F. The Reflective State Pattern. *Proceedings of the 5th Pattern Languages of Programs Conference (PLoP'98)*, August 1998, Monticello, Illinois, USA.

[7] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley, 1995.

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

[9] Lee, A., and Anderson, T. *Fault Tolerance: Principles and Practice,* Springer Verlag, 1990.

[10] Oliva, A., and Buzato, L.E. Composition of Meta-Objects in Guaraná, *Proceedings of the OOPSLA'98 Workshop: Reflective Programming in C++ and Java,* Vancouver, Canada, October 1998.

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

Resumo do Capítulo 2

Este capítulo apresentou dois artigos que documentam novos padrões de projeto. O primeiro artigo documentou um refinamento do padrão de projeto *State* utilizando uma arquitetura reflexiva. O principal objetivo é obter a separação dos aspectos de controle de uma máquina de estados dos aspectos funcionais da aplicação. O segundo artigo descreveu um sistema de padrões que propõem soluções para a implementação de técnicas de tolerância a falhas, definindo variações para o padrão *Reflective State*.

O padrão *Reflective State* é um padrão genérico que pode ser utilizado em contextos semelhantes ao do padrão *State* original. O sistema de padrões para o domínio de tolerância a falhas oferece mais detalhes de como a mesma estrutura do padrão *Reflective State* pode ser utilizada com diferentes semânticas para a implementação de diferentes técnicas de tolerância a falhas. Nós utilizamos a variação do padrão *Reflective State* para tolerância a falhas de ambiente no projeto e implementação dos componentes tolerantes a falhas do *framework* para controladores de trens. No capítulo 4 nós apresentamos com mais detalhes como este padrão foi aplicado no projeto detalhado do *framework*.

A implementação do padrão *Reflective State* genérico representa também um *framework* para a implementação de máquinas de estados. Este *framework* pode ser estendido através de configuração e/ou derivação, de acordo com as discussões sobre a implementação do padrão *Reflective State* apresentadas no primeiro artigo. A utilização da arquitetura reflexiva do Guaraná [Oli98] proporcionou uma estrutura de meta-nível facilmente reconfigurável, podendo-se inclusive obter a reconfiguração da máquina de estados em tempo de execução.

O próximo capítulo apresenta os estilos de arquiteturas que foram utilizados no projeto da arquitetura do *framework*.

Capítulo 3

Arquiteturas de Software e Estilos

A arquitetura de um sistema de software compreende os componentes computacionais e as interações entre estes componentes, definindo também a relação entre os requisitos e os elementos de software [SG96]. Um *estilo de arquitetura* define um padrão para a estruturação e organização geral de uma classe de sistemas. Um estilo define um vocabulário de componentes e conectores e um conjunto de restrições de como estes elementos de arquitetura podem ser combinados para definir a arquitetura de um sistema.

Na construção de sistemas de software complexos, que incluem vários requisitos não-funcionais tais como tolerância a falhas e distribuição, é essencial a escolha adequada de estilos de arquitetura que ofereçam soluções para manter a complexidade adicional sob controle. Em particular, no desenvolvimento do *framework* para controladores de trens, nós estávamos interessados em estilos de arquitetura que ajudassem na definição da estrutura geral do *framework*, considerando a implementação dos requisitos não-funcionais de tolerância a falhas. Dos estilos existentes, nós escolhemos o estilo *Meta-Level* para a implementação dos aspectos de gerência relacionados às técnicas de tolerância a falhas de uma forma separada e transparente para a aplicação. Para implementação da estruturação geral e o inter-relacionamento entre os componentes tolerantes a falhas, nós utilizamos o modelo de componente tolerante a falhas ideal proposto por Lee e Anderson, e o descrevemos como um estilo de arquitetura, contribuindo assim para sua reutilização.

Este capítulo é composto pelo artigo "*Architectural Styles and Patterns for Developing Dependable Frameworks*", que foi submetido para "*30th International Conference on Dependable Systems and Networks* (FTCS-30)", a ser realizado de 25 a 28 de junho de 2000, em Nova York, NY, Estados Unidos. Este artigo apresenta dois estilos de arquitetura que podem ser

utilizados no desenvolvimento de sistemas tolerantes falhas, com o principal objetivo de reduzir sua complexidade: (1) o estilo *Idealized Fault-Tolerant Component*, que é apresentado como uma proposta nova de estilo de arquitetura e (2) o estilo *Meta-Level*. Para demonstrar as vantagens da utilização destes estilos na descrição da arquitetura de sistemas tolerantes a falhas, nós os aplicamos na definição da arquitetura do framework para controladores de trens tolerantes a falhas. Nós mostramos também como esta descrição pode ser refinada no nível de projeto de classes, utilizando o padrão de projeto *Reflective State* apresentado no capítulo 2.

Architectural Styles and Patterns for Developing Dependable Frameworks

Luciane Lamour Ferreira
Institute of Computing, State University of Campinas
e-mail: 972311@dcc.unicamp.br

Cecília M. F. Rubira
Institute of Computing, State University of Campinas
e-mail: cmrubira@dcc.unicamp.br

Abstract

Dependable systems tend to be complex due to the incorporation of component redundancy to implement fault tolerance. As the size and complexity of these software systems increase, software developers have recognized the importance of exploiting design knowledge in the definition of their overall system structure, i.e. its software architecture, by means of reusing common patterns of system's organization. This kind of reuse can be achieved by using architectural styles and patterns, which provide well-proved solutions for common design problems. The aim of this paper is twofold. First, we identify two architectural styles that are important for the architectural descriptions of object-oriented dependable systems: (i) the new architectural style "*Idealized Fault-Tolerant Component*", which is based on exception handling mechanisms to separate the normal from the abnormal activities of interacting fault-tolerant components; and (ii) the *Meta-Level* architectural style, which divides an application into two levels, the meta and the base level, and applies the computational reflection concept to allow meta-level objects to change structure and behavior of base-level objects. These architectural styles aim to reduce the complexity of dependable systems and to improve our understanding of their software architecture. Second, we show how both architectural styles and existing design patterns can be used in the development of an object-oriented dependable framework for train controllers.

Keywords: fault tolerance, exception handling, architectural styles, computational reflection, framework.

1 Introduction

Modern software systems have the important requirement for dependability. In order to achieve dependability despite the presence of faults, measures for fault tolerance should be adopted [LA90]. In general, techniques for achieving fault tolerance depend upon the effective deployment and use of component redundancy, what could lead to an increase in the system size and complexity. As the size and complexity of these software systems increase, software developers have recognized the importance of exploiting design knowledge and expertise in the engineering of new dependable systems. The reuse of design solutions for common problems can be achieved by reusing both *architectural styles* for the overall system organization, i.e. the system architecture, and *design patterns* for the detailed design of the system's components. In this context, we are interested in architectural styles and patterns that provide solutions for mastering the complexity of dependable systems.

The aim of this paper is twofold. First, we identify two architectural styles that are important for the architectural descriptions of dependable systems on the object-oriented paradigm, providing appropriate patterns of system's organization for the provision of fault tolerance. These architectural styles are: (i) the new architectural style "*Idealized Fault-Tolerant Component*", which is based on exception handling mechanisms to separate the normal and abnormal activities of the interacting fault-tolerant components; and (ii) the *Meta-Level* architectural style, which divides an application in two levels, the meta and the base level, and applies the computational reflection concept to allow meta-level objects to change structure and behavior of base-level objects. These architectural styles aim to reduce the complexity of dependable systems and to improve our understanding of their software architecture.

Second, we show how these architectural styles can be used to define the architecture of an object-oriented dependable framework for train controllers, which uses exception handling and component redundancy to implement fault tolerance in its architecture. These styles provide a solution for reducing the complexity of the framework's architecture, allowing a clear separation of the fault tolerance implementation from the implementation of its basic functionality. In this paper, we present the basic model of a specific application from the domain, and describe its architecture using the architectural styles. We also show an example of how these styles can be refined in the detailed class design using design patterns.

This paper is organized as follows. Section 2 introduces some concepts on software architecture, architectural styles and design patterns. Section 3 defines two architectural styles for developing dependable systems: the *Idealized Fault-Tolerant Component* style and the *Meta-Level* style. Section 4 presents a case study of developing a dependable framework for train controller applications, using these styles to define the architecture of the framework. We also show how they can be combined and refined in the class design level of the framework, using design patterns. Section 5 presents the conclusions of this paper.

2 Reuse Techniques

In this section, we present the concepts of software architecture, architectural styles and design patterns, which have been used to promote the reuse of good architectural and detailed design solutions. These concepts have been applied in the development process of a dependable framework for train controllers, which follows the steps presented in the Figure 1.

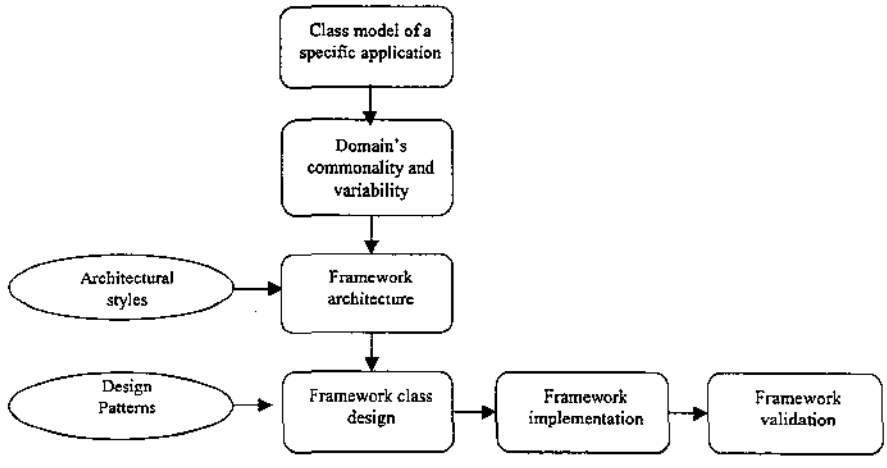


Figure 1: The framework development using architectural styles and design patterns

2.1 Software architecture and architectural styles

According to Shaw and Garlan [SG96], the "software architecture" of a software system can be described as the description of *elements* from which systems are built, *interactions* among those elements, *patterns* that guide their composition, and *constraints* on these patterns. In addition to specifying the structure and topology of the system, the architecture shows the correspondence between the system requirements and elements of the constructed system, thereby providing some *rationale* for the design decisions.

An important question on software architecture definition is how to leverage past experience on software architecture to produce better designs. Architectural structures are often described in terms of idiomatic patterns that are used informally by the system's architects, such as "client-server system", a "blackboard system", a "layered system", etc. These idiomatic patterns of system organization are defined as *architectural styles* [SG96]. They capture specific organization principles and structures for certain classes of software, and allow a shared understanding of the common forms that can be used by the architects. An architectural style defines a *vocabulary* of components and connectors, a set of *constraints* on how they can be combined and *semantic models* that specify how to determine a system's overall properties from the properties of its parts.

The use of architectural styles has a number of significant benefits [MKMG97]: (1) it promotes design reuse at the architectural level, where well-understood properties can be reapplied to new problems with confidence; (2) it can also lead to code reuse: often the invariant aspects of an architectural style lend themselves to shared implementation, for instance, in the client-server styles, one can take advantage of the RPC (remote procedure call) mechanism to implement the remote service invocations in a server; (3) it is easier for others to understand a system's organization if conventionalized structures are used; (4) by constraining the design space, an architectural style often permits specialized, style-specific analyses.

A style can be defined answering the following questions [SG96]:

1. What is the design vocabulary: types of components and connectors?
2. What are the allowable structural patterns, i.e. the design rules?
3. What is the underlying computational model?
4. What are the essential invariants of the style?
5. What are some common examples of its usage?
6. What are the advantages and disadvantages of using that style?
7. What are some of the common specializations?

2.2 Design patterns

A design pattern names, abstracts and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design [GHJV95]. A design pattern identifies the participating classes and instances, their roles and collaborations, and the

distribution of responsibilities. Compared to architectural patters, design patterns refine the general components of an architectural style, providing the detailed design solutions. Usually, the selection of a design pattern at the detailed design phase is influenced by the architectural styles that were previously chosen at the high-level design phase.

3 Architectural Styles for Dependable Systems

3.1 The Idealized Fault-Tolerant Component Architectural Style

According to Lee and Anderson [LA90], a system can be viewed as a set of components interacting under the control of a design (that is itself a component of the system). The system model is recursive in the sense that each component can itself be considered as a system on its right, and thus can have a recursive structure composition which identifies further sub-components. Moreover, these components receive requests for service and produce responses. If a component cannot satisfy a request for service, then it will return an exception. At each level of the system, an idealized fault-tolerant component will either deal with exceptional responses raised by components at a lower level or else propagate the exception to a higher level of the system (Figure 2).

In order to obtain dependability, each interacting component of the system should be dependable, i.e., each component should perform its job according to the specification and should be capable to handle abnormal situations (caused either by its own computation or by computations of the components with which it interacts). The *Idealized Fault-Tolerant Component* style is defined below according to the questions presented in the section 2.1.

3.1.1 What is the design vocabulary?

The vocabulary of the design elements consists of (i) components and (ii) connectors types that can be used by the system architects to draw the architectural diagram. The possible components of our style are the idealized components that can play two different roles (usually a component plays both roles at the same time):

- Idealized-supplier component: it receives requests for services and produces responses. If the service is performed according to the specification, the component returns a normal response, otherwise it returns an abnormal or exceptional response. The normal and abnormal

responses are both part of the component's interface so that a client can provide means to handle the component's responses.

- Idealized-client component: it requests services to the supplier components. The client should be responsible for providing means to handle the normal and especially the abnormal responses from idealized-supplier components.

The connectors of a style define the way that the components communicate with each other. In the *Idealized Fault-Tolerant Component* style, there are only two forms of communication between components: (1) service requests and (2) service responses. A request can be represented as a procedure call, a message, an event sent to a component, and so on. The components interact with each others by means of their public interface. The public interface is "any place" of interaction between two components, which has been previously set by the interacting components. For instance, using this style combined with the object-oriented model, a service request can be a method invocation sent to an object. Moreover, normal and abnormal responses are part of the method declaration defined in the public interface of the object's class.

3.1.2 What are the allowable structural patterns, i.e. the design rules?

Figure 2 shows the structural organization of the *Idealized Fault-Tolerant Component* style. An *idealized component* is divided in two parts, the normal part that implements its normal activities, and the abnormal part that implements the measures for tolerating faults that cause exceptional responses. In an ideal situation, both client and supplier components interact with each other producing only normal responses. However, considering that a system is not free from faults, exceptions may be produced as responses of client-component requests that cannot be satisfied due to supplier-component faults. More specifically, a component fault is an error in the internal state of a component whereas a design fault is an error in the state of the design.

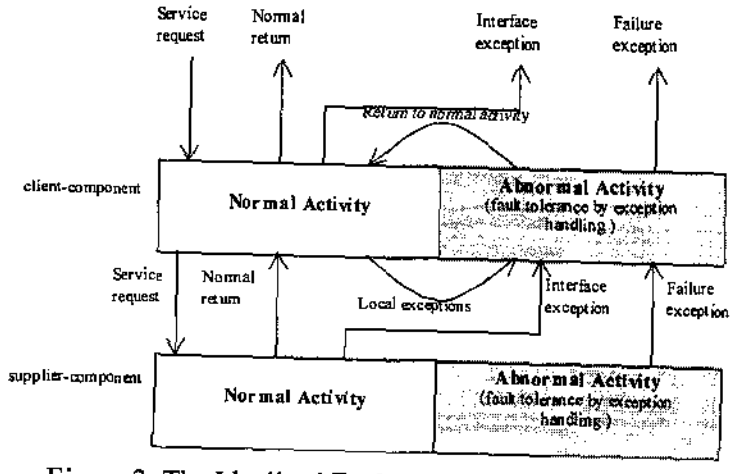


Figure 2: The Idealized Fault-Tolerant Component style

The abnormal responses (exceptions) that can be returned as response of a service request that cannot be satisfied are:

- Interface exceptions: are signaled in response to a request which did not conform to the component's specified interface. For instance, a parameter value is not in a specific range. This means that the preconditions of a service were not satisfied by a client-component.
- Local or internal exceptions: are exceptions generated by the component in order to invoke its own internal exception handlers.
- Failure exceptions: are signaled if a component determines that for some reason it cannot provides its specified service.

In the *Idealized Fault-Tolerant Component* style, interface exceptions are signaled in the normal part of the component, while failure and interface exceptions from supplier components invoke the exception handling part of the client component. If these exceptions are handled successfully (that is, the component was able to mask the exception), the component can return to providing normal services. However, if the component does not succeed in dealing with such exceptions, it should signal a failure exception to a higher level of the system.

3.1.3 What is the underlying computational model?

The style does not restrict the underlying computational model, since it is based on the composition of a system by generic components that can be of many types. A component can represent, for instance, a module in procedural languages, a class or package in object-oriented systems, a data repository, a hardware device, an environmental entity, and so on.

that a component can get information about the internal properties of another component and based on that information it can dynamically interfere on its current computations.

For the meta-level to be able to reflect on base-level objects, it must be given information regarding the internal structure of base-level objects (structural meta-information). The representation of abstract language concepts such as classes and methods, in form of objects, is called *reification*. Moreover, interaction between objects may also be materialized as objects, so that meta-level objects can inspect and possibly alter them. This is achieved by *intercepting* base-level operations such as method invocations, creating objects that represent them, and transferring control to the meta-level. After transferring the control to the meta-level, the meta-objects can inspect the reified information and can also modify structural and/or behavioral aspects of the base-level object. This process is also called *reflecting* the changes back into the base-level object [Mae87, Fer89].

The connector of this style is the meta-object protocol (MOP) [KRB91], which establishes the relationship among the base-level and meta-level objects. The MOP provides an interface to the programming language implementation in order to reveal to the program information normally hidden by the compiler and/or run-time environment [Mae87]. The various existent MOPs differ mainly in the following features: (1) binding between base-level objects and meta-level objects, which can occur staticaly (at compile-time)[GR89] or dynamically (at runtime) [Oli98]; (2) class-wide reflection (each class is associated with a single meta-class) [GR89, Fer89] and object-wide reflection, where objects are attached to meta-objects [YTT89, GC96, Oli98]; (3) cardinality of the relationship between base-level objects and meta-level objects.

Figure 3 shows the structural organization of a system that uses the *Meta-Level* architectural style. The interactions between base-level and meta-level objects are realized through a meta-object protocol which establishes the allowable design rules that guide the construction of a system organized with this style. Each specific MOP gives a different structural organization for a system that is implemented using it.

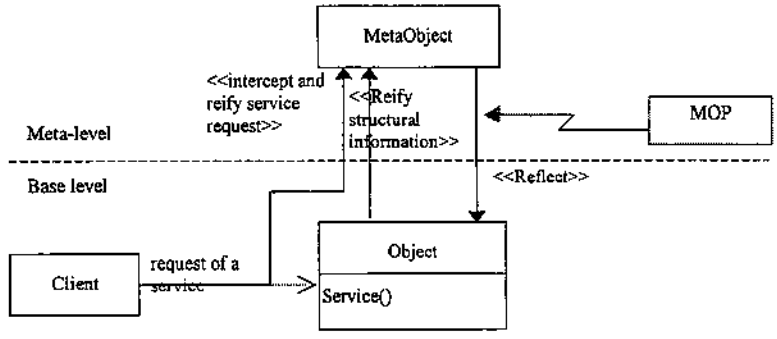


Figure 3: The Meta-Level architectural style

Meta-level architectures address separation of concerns, providing means to implement non-functional properties of an application transparently separated from its functional properties. The functional requirements are primarily concerned with the purpose of an application, while the non-functional requirements are more concerned with its fitness for purpose. The meta-objects usually are responsible for implementing non-functional requirements such as fault tolerance [BRL97, Lis98, FPB95], persistence [SW94], distribution [Str92, YTT89, Oli98], etc.

The object-oriented meta-level style has already been detailed documented as an architectural pattern named *Reflection* pattern by Buschmann et.al. [BMRS+96]. We have described only the main features of this style that are necessary to understand its applicability to define the architecture of dependable systems. For more details about this style, the reader should refer to [BMRS+96].

4 A Case Study: a Dependable Object-Oriented Framework for Train Controllers

In order to illustrate the use of the *Idealized Fault-Tolerant Component* style and the *Meta-Level* style, we describe a case study of an object-oriented dependable framework for train controllers. The framework for train controllers is a generic software system that should encompass the common functionality of a related family of train controller applications, aiming to provide reusability in large scale. The framework implements the common requirements (both functional and non-functional) of train controllers in its architecture, and provides adaptable parts that should be extended or configured to accomplish application-specific requirements. Our goal is to demonstrate how these two architectural styles can be applied to describe the framework's architecture so that its dependability property is achieved. We also show how the framework

architecture can be refined until the detailed design level, using design patterns to refine the general architectural components.

As illustrated in Figure 1 (Section 1), the first step of the framework development is the analysis of a specific application from the domain. In our case study, we limit our domain to a subdomain of a railway model: a simplified model of controlling and monitoring system for a train set. Our start point is a specific software that was developed earlier to control this simplified railway model, called the Train Set System [Rubira94, Quadros97]. The next section describes the main requirements of this application.

4.1 The Train Set System

The Train Set System is a digitally controlled model railway, which is divided into three parts: electronic digital units, railway layout and trains (Figure 4). The railway layout is mounted on separated boards that can be independently controlled by separated controllers. Each board can be viewed as being composed of a set of switches, sensors and railway tracks, which link connectors and sensors (which we call stations, since they are the only source of information about the state of the system). Sections are directed links of railway tracks between adjacent stations, and can contain a sequence of zero or more connectors. The sections are abstractions used by the trains to move around the boards.

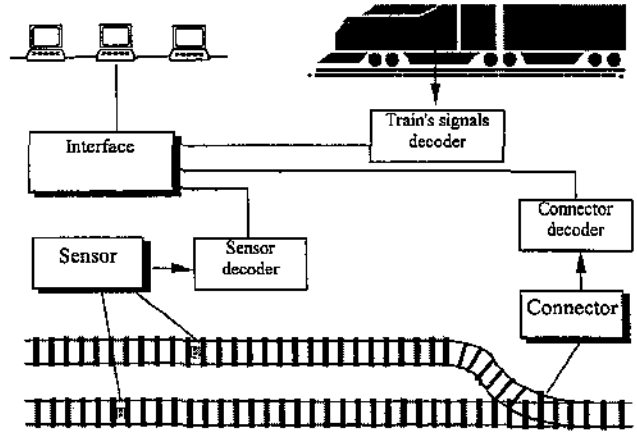


Figure 4: The Train Set system and the representation of the Marklin hardware

The trains aim to move randomly between stations. The switches and sensors are unreliable devices, and can suffer environmental faults. Despite the presence of faulty switches and sensors, the trains should move around the railway without crashing, but if necessary stopping and reversing. The railway layout is divided up in three boards, and the design solution should take

into account the layout distribution and the train crossings between neighbor boards. The relevant restrictions of such application can be summarized as following:

- The main goal is to guarantee no train collision, i.e., safety;
- Switches and sensors are unreliable devices and can suffer environmental faults. However, considering two consecutive sensors it is assumed that only one sensor can fail.
- Derailment of train is not considered.
- Routing of the train is ignored, i.e., the trains move randomly between stations.
- Train can stop within one section that means that it travels slowly enough to stop immediately when requested.
- We assume that the train size is smaller than the smallest section and can be completely contained within a section.

4.2 Architectural description using the Idealized Fault-Tolerant Component style

Following the *Idealized Fault-Tolerant Component* style, we identify the main dependable components of the train controller system and their most important interactions at the architectural level (Figure 5). Some important aspects that should be highlighted are: what are the most essential interactions between the main dependable components; what are the possible exceptions that can be returned; how the exceptions are propagated and handled; what components are responsible for the error recovery.

At a higher level of abstraction, we identify three main dependable components that interact with each other to provide the main functionalities of the system: Train, Controller and Board. These components can be represented as packages in an object-oriented system which encapsulate details of their classes. The packages provide public services that are in turn provided by their constituent classes.

The Train component is responsible for the movement of trains, implementing services such as move(), stop(), revert(), etc. The Controller component is the central part of the system, and it is the intermediate component between the Train and the Board. The Controller provides important services to the trains so that they move around the boards without crashing, such as lock_section(), release_section(), etc. The Board component defines the detailed representation of

the board's layout, and is composed by a set of connectors, sensors and sections. It is responsible for creating, initializing and managing these components.

As stated in the section 4.1, the goal of the system is to control the movement of trains so that no train collision occurs. For that, a Train requests services to the Controller, such as lock_section(), release_section() and occupy_section(). If the service is executed normally, the Train receives a normal response and continues moving toward the next section. If some fault occurs in a supplier-component, the service request cannot be performed normally and an exception is signaled and propagated to the Train. For instance, the Board can return an exception as a response of the operation lock_section() due to some faulty sub-component (e.g. a switch that belongs to the required section is faulty and the section cannot be used by a Train). The exception is propagated through the components that are between the lower-level component, i. e. the switch device, and the higher-level component, i.e. the Train. The Train is the component responsible for handling the exception, using a forward error recovery mechanism: if a requested section is abnormal, the train attempts to lock another one in order to recovery from the error.

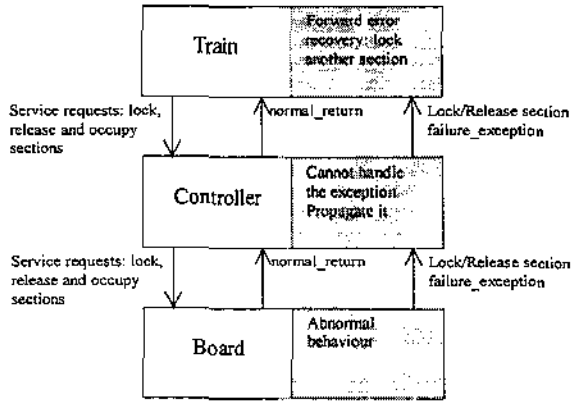


Figure 5: The architecture of the dependable framework for train controllers using the Idealized Fault-Tolerant Component style

This architecture description can also be refined hierarchically by decomposing the components into sub-components. The Board component is composed by the Section sub-component, which in turn is composed by the Switch sub-component. Figure 6 shows how the components and sub-components collaborate to execute the requested services. The exceptions are propagated from the lower-level component Switch until the higher-level component Train. The type of the propagated exception is different for each level of abstraction. For instance, a faulty Switch can signal the exception "switch_abnormal_failure_exception", and this exception is propagated as a "lock_failure_exception" by the Section, and as a "lock_section_failure_exception" by

the Board. For the Train is interesting to know only that the Section cannot be locked. This description can be incrementally refined until the design level. At the detailed design level, the components should be decomposed in smaller design elements and the possible interactions between them should be identified and described using a detailed event sequence diagram.

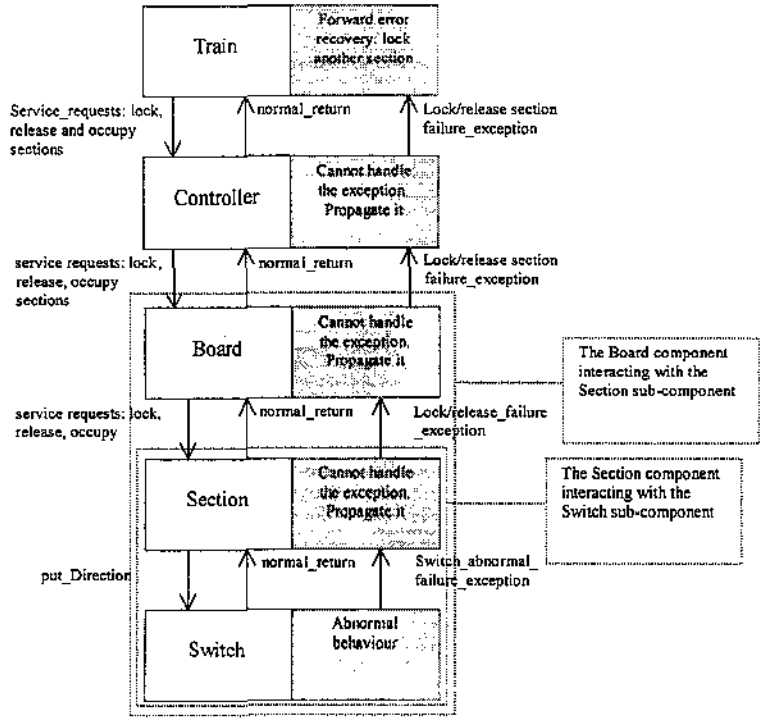


Figure 6: The architecture description with more details: the component Board was decomposed into two sub-components.

From this architecture description, we can identify some important aspects of the train controller domain:

- The dependencies between the components: the architecture description shows how the components depend on the dependable services of each other. The Train depends on the dependable services of the Controller that depends on the dependable services of the Board, and so on. It means that the Train component should know only the public interface of the Controller component, and should provide handling for the exceptions signaled by it, no matter if the exception was signaled first in a lower-level component. The capacity of propagating and resignalling different exceptions allows the definition of exceptions at different level of abstractions.
- It is possible to identify the source of faults (in this case, the Switch component) and which component is responsible for performing the error recovering. The exception generated by a

faulty component (an abnormal switch) is propagated until the Train component that implements the handler for this kind of exception.

4.3 *Architectural description using the Meta-Level style*

In this section, we describe the architecture of the dependable framework for train controllers using the *Meta-Level* architectural style, emphasising the aspects related to the fault tolerance implementation. The main aspects to be considered are: (i) the identification of the components that belong to the base level and meta-level; (ii) the responsibilities of each component and (iii) how they interact using a specific meta-object protocol (MOP). Furthermore, we can also identify the impact of using this style on the main "quality" features of the system, such as the degree of adaptability that can be obtained by means of the reflection mechanisms and the impact of using the reflection mechanisms on the system's performance. Figure 7 shows the main components and their relationships.

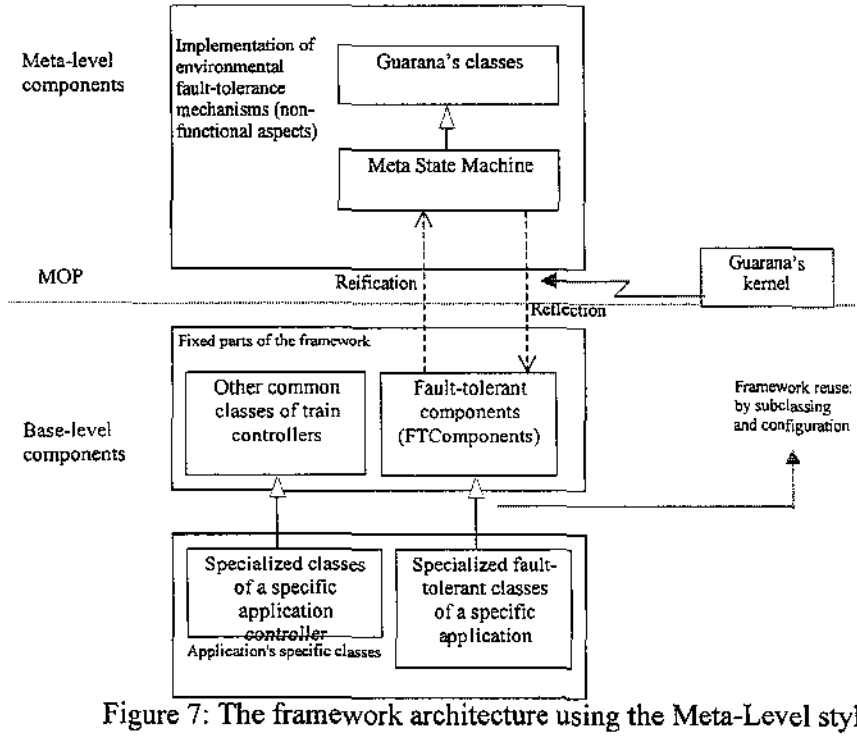


Figure 7: The framework architecture using the Meta-Level style.

The meta-level encompasses the components responsible for implementing the non-functional requirements of the application. In the framework for train controllers, one important non-functional requirement is the control of the redundant components used to implement environmental fault-tolerance. The entities of the environment may exhibit different behavior phases during their lifetime. For instance, if a switch of the environment is faulty, the switch

object (in the solution domain) should present a different behavior for the execution of its methods. The approach used to implement environmental fault tolerance consists on defining state classes that encapsulate the different implementation for the component's services [Rub94]. The component changes its behavior by changing its current state object.

The aspects related to the control of the state objects and the state transition execution are implemented by the MetaStateMachine at the meta-level. The MetaStateMachine is responsible for maintaining the information about the possible states and the current state of the component, for changing its current state, and for delegating the state-dependent services to the correspondent state object. The MetaStateMachine is configured at runtime accordingly to the state diagram definition of a FTComponent. The association of the MetaStateMachine and the FTComponent at the base level is dynamic.

We have used the Guarana's MOP [Oli98] to implement the coupling between the meta and base-level components. The Guarana's reflective architecture also defines a framework that provides the base classes for implementing the basic meta-level behavior. The MetaStateMachine reuses this framework by inheriting basic classes such as MetaObject and Composer. It overrides some methods of these classes to implement the specific meta-level behavior of the MetaStateMachine.

The interactions between the MetaStateMachine and the FTComponents are performed transparently by means of the interception and reification mechanisms provided by the Guarana's kernel, enforcing the separation of concerns. The MetaStateMachine inspects the entire operations target to its associated FTComponent, performing the control aspects of the state machine execution.

An important advantage of using the *Meta-Level* style is the capacity of constructing systems that are easily adaptable to new requirements. In the framework for train controllers, the FTComponents can be extended to accomplish application's specific requirements, and their state diagram can also be extended or modified. Using this style, the state machine implementation can be easily extended by modifying the configuration of the MetaStateMachine, without affecting the service's implementation of the FTComponent and its correspondent state classes. This solution separates the changes in the specific state machine configuration from the changes in the functional services of the application.

As we have stated before, the *Meta-Level* architectural style also has some disadvantages. In order to implement the state machine control in the meta-level, many extra method invocations are necessary due to the interception and reification mechanisms. It causes an impact on the system performance. Furthermore, the Guarana's kernel intercepts the operations target to a base-level object, and many operations are intercepted unnecessarily, since it is not possible to select only those we are interested.

4.4 The class design of the framework using design patterns

The previous sections have described the architecture of the framework for train controllers using both the *Idealized Fault-Tolerant Component* style and the *Meta-Level* style. In this section, we describe how these two styles influence the class design of the dependable components. At the class design level, the components are refined using design patterns that follow the general structures of the architectural styles that were chosen before.

To illustrate the class design phase, we refine the idealized component Switch defined in the section 4.2, figure 6. This is a lower-level idealized fault-tolerant component that represents a real switch of the environment. The normal and abnormal part of the Switch component represents the normal and abnormal behavior of a real switch of the environment, since a switch is a possible source of environmental faults. The normal part executes the services returning normal responses, while the abnormal part returns failure exceptions as responses for its services, since a faulty switch cannot provide its services normally.

The design of the Switch component follows the *Reflective State* pattern [FR98a] [FR98b], which is based on the *Meta-Level* style. This pattern encapsulates the normal and abnormal behavior of a component in separated state classes defined at the base-level, and defines a MetaStateMachine at the meta-level, which is responsible for implementing the control aspects related to the state transition execution, as we have discussed in the previous section. Figure 8 shows the design of the Switch component following this pattern. The Switch and SwitchNormalState classes implement the normal part of the Switch component, and the SwitchAbnormalState class implements the abnormal part. The MetaStateMachine controls the transition from the normal to the abnormal states of the Switch component, and holds the reference for the current state object. The transition to the abnormal state is triggered when the correspondent switch of the environment is faulty. After changing to the abnormal state, any

service request sent to this Switch component will be handled by its abnormal part. The MetaStateMachine also performs the delegation of the state-dependent service to the current state object. The methods of the SwitchAbnormalState class return failure exceptions indicating that the component cannot provide its services normally. The exception is propagated until it reaches the higher-level component, as we have showed before in the section 4.2.

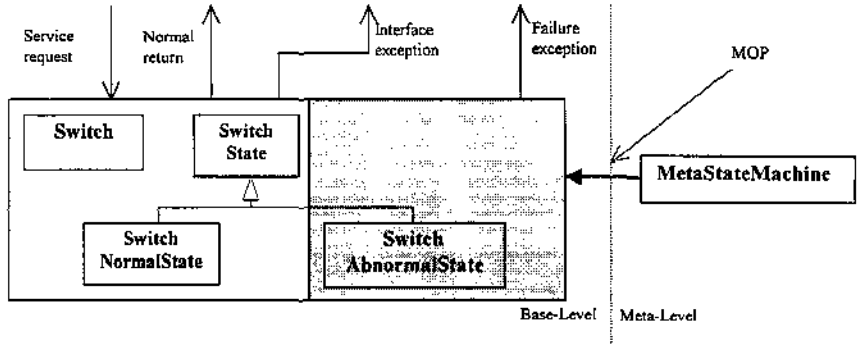


Figure 8: The class design of the idealized fault-tolerant component Switch using the Reflective State pattern.

We have implemented the exception handling and exception propagation using the exception handling mechanism provided by the Java programming language. In fact, the normal and abnormal part (responsible for the exception handling) of the higher-level components are not clearly separated. The exception handling is implemented using the "try" and "catch" commands of Java, which are implemented within the method's body. Another possible solution is to implement the exception handling mechanism using the *Meta-Level* style, as proposed in [GBR99]. This work defines the exception handlers in a separated class at the base level and implements the control of the exception handling mechanism at the meta-level. The meta-level objects are responsible for the following activities: (i) search for a suitable handler associated to the raised exception; (ii) invocation of the handler; (iii) return to the normal operation of the application.

4.5 Implementation issues

The dependable framework for train controllers has been implemented using the Java programming language and a meta-object protocol called Guaraná [Oli98] to implement the MetaStateMachine at the meta-level. The whole framework's implementation has approximately 110 classes and 7000 lines of codes. The development of the framework is limited to the software controller itself, since we do not have access to the original Marklin hardware (the

digital units, the railway layout and the train engine) which was used to develop the Train Set System. In order to validate the framework, we have implemented and tested a specific application that reuses the framework and implements a simple simulator for the hardware sinals and for the faulty environmental components (switches and sensors).

5 Conclusions

We have discussed the benefits gained by reusing design knowledge and expertise in the definition of the software architecture for developing dependable systems by means of architectural styles. We have presented two architectural styles that provide appropriate patterns of system's organization for the design and provision of fault tolerance: (i) the "*Idealized Fault-Tolerant Component*" style and (ii) the *Meta-Level* style. The first one is based on exception handling mechanisms to separate the normal and abnormal activities of the interacting fault-tolerant components. The second one applies meta-level programming to separate the system into two different levels and a meta-object protocol to implement the interactions between the two levels. This style can be applied to define the software architecture of dependable systems to provide the separation of non-functional properties related to the provision of fault-tolerance from the functional properties in a transparent way.

We have also demonstrated the applicability of these two architectural styles in the development of dependable systems through a case study: the development of a dependable framework for train controller applications. We have shown how the styles can be applied to define the architecture of the dependable framework for train controllers, and how they can be refined at the class design level using design patterns.

The use of the *Idealized Fault-Tolerant Component* and the *Meta-Level* styles improves the understanding of the software architecture and guides the subsequent phases of the software development, that is, the detailed design, implementation, test and maintenance phases. Regarding our specific case study of the dependable framework for train controllers, these two architectural style have also improved the reusability of the framework, since they lead to a more understandable and well structured architecture design.

The use of the *Reflective State* pattern is one of the possible refinements of these two styles at the detailed design level. We can also use other design solutions that combine the *Idealized Fault-Tolerant Component* and the *Meta-Level* styles in different ways. As a future

work, we plan to implement the exception handling of fault-tolerant components using the reflective exception handling model proposed in [GBR99]. The idea is to implement the management activities related to the fault tolerance provision completely at the meta-level, transparently separated from the functional activities of the system's components.

6 Bibliography

[BMRS+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *A System of patterns: Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996.

[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, November 1997.

[Fer89] J. Ferber. Computational Reflection in Class-Based Object-Oriented Languages. *OOPSLA'89.* Vol. 24 no. 10, October 1989.

[FPB95] J.-C. Fabre, T. Pérennou, and L. Blain. Meta-object Protocols for Implementing Reliable and Secure Distributed Applications. *Technical Report LASS-95037,* Centre National de la Recherche Scientifique, February 1995.

[FR98] L. L. Ferreira and C. M. F. Rubira, The Reflective State Pattern. *Proceedings of the 5th Pattern Languages of Programs Conference (PLoP'98)*, August 1998, Monticello, Illinois, USA. Technical report # WUCS-98-25.

[GBR99] A.F. Garcia, D.M.Beder, C.M.F.Rubira. An Exception Handling Mechanism for Developing Dependable Object-Oriented Software Based on a Meta-Level Approach. To appear in *IEEE 10th International Symposium on Software Reliability Engineering*, Boca Raton, Florida, November, 1999.

[GC96] B. Gowing and V. Cahill. Meta-object protocols for C++: The Iguana Approach. In *Proceedings of Reflection '96,* pages 137-152. San Francisco, USA. April 1996.

[GR89] A. Goldberg and D. Robson. *Smaltalk-80: The Language.* Addison-Wesley, 1989.

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

[KRB91] G. Kiczales, J. des Rivieres and D. Bobrow. *The art of Meta-object Protocol.* MIT Press, 1991.

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

[Lis98] M.L.Lisboa. A New Trend on the Development of Fault-Tolerant Applications: Software Meta-Level Architectures. *Proceedings of the 1998 IFIP – International Workshop on Dependable Computing and its Applications.* Johannesburg, South Africa. January 1998.

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

[MKMG97] R. Monroe, A. Kompanek, R. Melton, D. Garlan. Architectural Styles, Design Patterns and Objects. *IEEE Software*, 14 (1): 43-52. January 1997.

[Oli98] A. Oliva. *Guaraná: Uma Arquitetura de Software para Reflexão Computacional Implementada em Java.* Master Thesis. Institute of Computing, State University of Campinas, September 1998. http://www.dcc.unicamp.br/~oliva/guarana/

[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, October 1994.

[RX93] B. Randell and J.Xu. Object-Oriented Software Fault Tolerance: Framework, Reuse and Design Diversity. *PDCS2 First Year Report,* Predictably Dependable Computing Systems, 1: 165-184, Toulose, France. September 1993.

[SG96] M. Shaw and D. Garlan. Software Architecture, Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs, NJ, 1996.

[Str92] R. Stroud. Transparency and Reflection in Distributed Systems. In *5th European SIGOPS Workshop on Models and Paradigms for Distributed Systems Structuring*, Mont Saint-Michel, France, September 1992. ACM SIGOPS, IRISA, INRIA-Rennes.

[SW94] R. J. Stroud and Z. Wu. Using Meta-Objects to Adapt a Persistent Object System to Meet Applications needs. *In 6th SIGOPS European Workshop on Matching Operating Systems to Applications Needs*. 1994.

[YTT89] Y. Yokote, F. Teraoka and M. Tokoro. A Reflective Architecture for an Object-Oriented Distributed System. In *Proceedings of ECOOP'89*, 1989.

Resumo do Capítulo 3

Este capítulo apresentou dois estilos de arquitetura que podem ser usados na definição da arquitetura de software de sistemas tolerantes a falhas: o estilo *Idealized Fault-Tolerant Component* e o estilo *Meta-Level*. Estes estilos foram utilizados na definição da arquitetura do *framework* para controladores de trens tolerantes a falhas. Nós mostramos também como estes estilos influenciam nas demais fases do seu desenvolvimento. O refinamento da arquitetura do *framework* através do projeto de classes mostrou como os estilos podem ser combinados no projeto detalhado, utilizando-se padrões de projeto.

Através deste projeto prático, podemos concluir que os estilos apresentados oferecem um padrão de estruturação adequado para a descrição da arquitetura de sistemas tolerantes a falhas, contribuindo para redução da complexidade de tais sistemas. Podemos concluir também que a escolha adequada dos estilos de arquitetura é muito importante para a obtenção de um projeto de boa qualidade.

Trabalhos relacionados:

Embora a documentação do estilo "*Idealized Fault-Tolerant Component*" no formato em que apresentamos seja nova, o modelo originalmente proposto por Lee e Anderson vem sendo utilizado há bastante tempo na estruturação de sistemas tolerantes a falhas. Como exemplo de utilização deste modelo, podemos citar o trabalho de Randell e Xu [RX93] que implementa tolerância a falhas de software baseando-se neste modelo. Este trabalho implementa redundância de software através de diversidade de projeto, definindo um componente tolerante a falhas ideal formado por um conjunto de variantes (que são também subcomponentes tolerantes a falhas ideais) e um árbitro. O componente retorna um resultado normal se o conjunto de variantes executar o serviço de forma correta de acordo com as condições de verificação definidas no árbitro. Caso contrário, o componente retorna uma exceção de falha. A abordagem deste trabalho é semelhante à nossa abordagem para implementação de tolerância a falhas de ambiente que utiliza o padrão *Reflective State* para refinar o projeto do componente tolerante a falhas ideal. A

principal diferença refere-se à forma como uma resposta normal ou anormal é produzida: nós utilizamos objetos de estado para implementar a parte normal e a parte anormal (que irá retornar exceções de falha), enquanto que o trabalho deles utiliza variantes de projeto e um árbrito para definir quando retornar um resultado normal e um resultado anormal que indica que o componente falhou na execução de um serviço.

Em relação ao estilo de arquitetura *Meta-Level*, existem vários trabalhos na literatura que propõem soluções reflexivas para implementação de requisitos não-funcionais, tais como distribuição [Str92, YTT89], persistência[SW94], etc. Em relação à tolerância a falhas, podemos citar os trabalhos [BRL97, Lis98, FPB95]. Todos estes trabalhos propõem soluções que visam separar os aspectos de gerência da implementação dos mecanismos de tolerância a falhas dos aspectos funcionais da aplicação.

No próximo capítulo nós apresentamos o projeto completo do *framework*, utilizando padrões de projeto e metapadrões para a descrição detalhada dos seus principais componentes, enfatizando-se a descrição dos pontos adaptáveis.

Capítulo 4

Projeto e Implementação de um *Framework* para Controladores de Trens

Nos capítulos anteriores, nós discutimos as técnicas de padrões de projeto e estilos de arquitetura, as quais são utilizadas principalmente para a obtenção de reutilização de soluções de projeto. Além disto, estas técnicas também auxiliam na documentação do projeto de *frameworks*, tanto no projeto da sua parte fixa como no projeto das partes adaptáveis.

A melhor forma de entender e analisar a efetividade destas técnicas é utilizá-las em uma aplicação prática. Neste capítulo nós apresentamos o projeto completo do *framework* para controladores de trens tolerantes a falhas e distribuídos. Na construção do *framework*, nós utilizamos os estilos de arquitetura e padrões de projeto que foram propostos nos capítulos anteriores, assim como outros padrões de projeto existentes na literatura. O projeto do *framework* é descrito no artigo "*The Design and Implementation of a Dependable and Distributed Framework for Train Controllers*", que foi submetido para a revista "*Software Practice & Experience*".

The Design and Implementation of a Dependable and Distributed Framework for Train Controllers

Luciane Lamour Ferreira
Institute of Computing, State University of Campinas
e-mail: 972311@dcc.unicamp.br

Cecília M. F. Rubira
Institute of Computing, State University of Campinas
e-mail: cmrubira@dcc.unicamp.br

Abstract

Object-oriented frameworks have emerged as a promising technique that allows large-scale reuse, providing reusability at the design and code levels. However, the development of frameworks is a complex and costly task since they should implement the application domain's commonalities and provide the appropriated adaptability for the specific features of each application. Therefore, it is necessary to apply effective design techniques to support its development. This paper presents the design and implementation of a dependable and distributed framework for train controller applications, and discusses the main design techniques used to support its development. Our main goal is to demonstrate the usefulness and limitations of applying techniques such as architectural styles, design patterns and metapatterns in the documentation of a complex framework, which includes various features such as fault tolerance and distribution.

Keywords: frameworks, architectural styles, design patterns and metapatterns.

1 Introduction

Recently, object-oriented frameworks have emerged as a promising technology for providing large-scale reuse, reducing the cost of software development and improving the quality of software. An object-oriented framework is a reusable, semi-complete application that can be specialized to produce customized applications. It provides a skeleton of an application, including the application's logic and flow of control, and allows the application developers to reuse not only the code but also the high-level design. Frameworks provide an easy way of developing new applications, however the development of such generic software is a complex and costly task. Therefore, it is necessary to apply effective design techniques to support its

development. Architectural styles [SG96], design patterns [GHJV95] and metapatterns [Pre95] have been proposed as a means of capturing and describing the design of frameworks.

In this paper, we describe the development of a dependable and distributed framework for the train controllers domain. The main goal is to provide a highly adaptable and understandable architecture that encompasses the main features of the domain of train controller applications, including features such as fault tolerance and distribution. The framework provides large-scale reuse, and it can be used in many different contexts, for instance, it can be extended to implement (i) the control of different railway models that control different kinds of trains; (ii) automatic fault diagnosis for the environmental faults (faults that occur in the components with which the controller interacts, such as switches and sensors), etc.

Our approach for the framework development follows the steps (Figure 1): (1) analysis and class design of a specific application from the framework's domain; (2) analysis and specification of the domain variability and adaptability; (3) high-level design of the fixed part of the framework, using architectural styles to define its architecture; (4) detailed design of the adaptable parts, which is defined by means of a sequence of generalizing transformation on the basic model of the step 1 applying design patterns and metapatterns for achieving the desired variability; (5) implementation of the framework using a specific programming language; (6) validation of the framework by reusing it to implement specific applications.

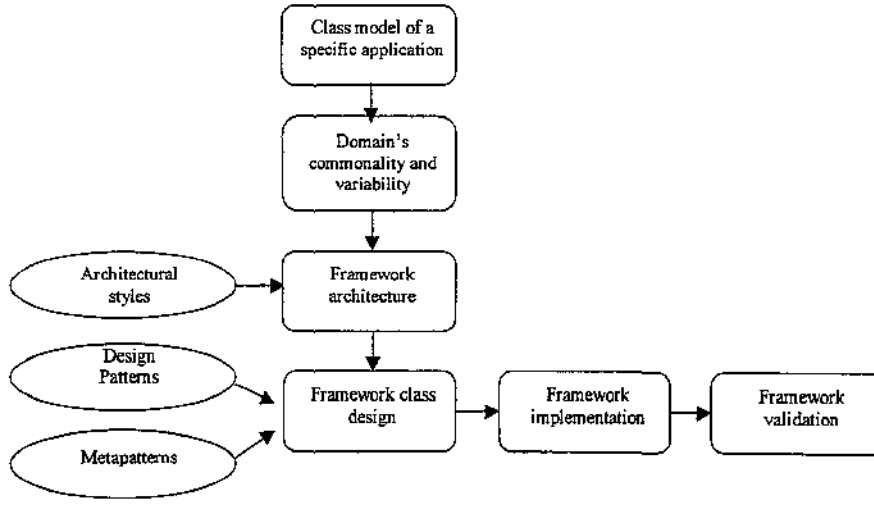


Figure 1: The framework development using architectural styles and design patterns

We apply the hot-spot-driven approach to define the adaptable parts of the framework [Pre95, Sch97]. Hot spots are the aspects of the application domain that should be kept flexible

while frozen spots represent the common aspects of the domain that cannot be changed. Design patterns and metapatterns are applied to describe the framework's hot spots. Architectural styles are applied to describe the framework's architecture, defining the framework's fixed structure and flow of control. Our main goal is to demonstrate the usefulness of applying these techniques in the development of this complex framework. We show how these techniques have improved the framework understanding, providing documentation for both the framework reuse (in the development of new applications) and the framework's maintenance and evolution. We also discuss the main lessons learned in the development of this complex application, and present our conclusions about the advantages and the limitations of using architectural styles, design patterns and metapatterns for documenting the framework.

This paper is organized as follows. Section 2 introduces some reuse techniques that have been used recently to achieve large-scale reuse, such as frameworks, architectural styles, design patterns and metapatterns. Section 3 describes the basic model of a specific train controller application, which has been used as the start point to develop the framework. Section 4 describes the framework design, identifying the frozen and the hot spots and documenting them using architectural styles, design pattern and metapatterns. Section 5 presents the conclusions of the paper, discussing the main advantages and limitations of using patterns to document the framework design.

2 Reuse Techniques

2.1 Object-Oriented Frameworks

Framework is an emerging object-oriented reuse technique that has been used by software developers to increase the productivity of software development by using effective large-scale reuse. As stated by Halph Johnson [Joh97a, Joh97b], a framework is a reusable design of all part of a system that is represented by a set of abstract classes and the way their instances interact. Its purpose is to provide the skeleton of an application that can be customized by an application developer. Frameworks are in the middle of the reuse techniques, providing both code and high-level design reuse. It also allows the communication and sharing of designer's experience and domain expertise.

One of the characteristics of frameworks is inversion of control, which makes them different from traditional reuse techniques such as class library. A developer reuses components from a class library by writing its program that calls the library's components whenever necessary. The developer is responsible for implementing the overall structure and the flow of control of the application. Reusing a framework is different: the framework provides the overall structure and the flow of control, and the application developer implements only the parts that should be plugged into the framework. The framework code calls the developer's code [Joh97a]. This framework's feature lets developers reuse the application logic, reducing the cost and improving the quality of software.

A common approach used to the framework development is based on the so-called hot spots and frozen spots [Pre95, Sch97]. Framework's designers specify variations within the design by means of hot spots, which are those aspects of an application domain that have to be kept flexible; application's developers refine the framework design for the needs of their application by filling in those hot spots. A hot spot lets the developers "plug-in" an application-specific class or subsystem, either by selection from a set of those supplied with a *black-box framework*, or by programming a class or subsystem, usually by inheritance and dynamic binding, in a *white-box framework* [Sch97]. Usually, frameworks provide both black-box and white-box kind of adaptability. The frozen spots are the fixed part that provides the overall structure and application's logic. They define the framework's architecture in terms of its components and their relationships, the components' responsibilities and collaborations to perform the main functionalities of the application's family.

Frameworks evolve through its various uses by different applications. This evolution means that in its early stages, the framework is mainly conceived as a white-box framework. However, the framework matures through being adopted in an increasing number of applications. More concrete components providing black-box solution for problems found in the domain become available within the framework [BMA97].

2.2 Architectural styles

As stated in [SG96], architectural styles are idiomatic patterns of system organization. They capture specific organization principles and structures for certain classes of software, and allow a shared understanding of the common forms that can be used by the architects. An

architectural style defines a *vocabulary* of components and connectors, a set of *constraints* on how they can be combined and *semantic models* that specify how to determine a system's overall properties from the properties of its parts.

Regarding the framework development, architectural styles can be applied to define the framework's architecture, promoting design reuse at the architectural level. The use of architectural styles makes easier for others to understand the framework's organization, since they provide conventionalized and well-known structures. When the framework includes various features such as fault-tolerance and distribution, it is also important to make the appropriated choice of the architectural styles that provide good solutions for reducing the framework complexity and improving the framework's understanding.

2.3 Design patterns

Patterns have been used to support the reuse of software design. Their primary goal is to communicate good, well-proved and recurring design solutions for common software problems. As stated by Gamma et al. [GHJV95] a design pattern names, abstracts and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. A design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Patterns are useful not only for describing successful solutions, but also for improving the vocabulary among software developers, communicating information between designers, programmers and maintenance programmer at a higher level than individual classes or functions [Cli96].

In general, a pattern has four essential elements [GHJV95]: (1) The *pattern name* describes succinctly the design problem, its solutions and consequences using a word or two; (2) the *problem* describes when to apply the pattern, explaining the problem and its context, and presenting a list of conditions that must be met before it makes sense to apply the pattern (the pattern's forces); (3) the *solution* describes the elements that make up the design, their relationships, responsibilities and collaborations; (4) the *consequences* are the results and tradeoffs of applying the pattern.

Most of the design patterns from existent pattern catalogs [GHJV95, BMRS+96] provide solutions for variability and adaptability problems, defining a common vocabulary to document the framework's hot spots. However, these design patterns do not communicate precisely which

are the adaptable and fixed parts, and how to provide specific implementations for each hot spot. To solve this problem, Pree has proposed the use of metapatterns [Pre95] (which is also referred as a design pattern) as a means for documenting more precisely the framework's hot spots. We present the concepts of metapatterns in the next section.

2.4 Metapatterns

According to Pree [Pre95] metapatterns are defined as a set of design patterns that describe how to construct frameworks independent of a specific domain. These metapatterns can be applied to categorize and describe any framework example[2] design pattern on a meta-level and therefore, they are considered more abstract than state-of-the-art design patterns. Metapatterns do not replace these design patterns, but complement them. They are primarily used for documenting the framework's hot spots during the design process, improving the framework understanding.

Metapatterns are based on template and hook methods. *Template methods* implement the frozen spots and the *hook methods* implement the hot spots of a framework. The template methods are a means of defining abstract behavior or generic flow of control or the relationship between objects. A template method can be considered as a complex method that is implemented based on the elementary hook methods. The *hook methods* can be either: (i) abstract methods; (ii) regular methods or (iii) template methods. Figure 2 illustrates the concepts of template and hook methods. Method M1() is a template method that calls its hook methods M2() and M3(). Method M2() is an abstract method of B, and the subclass B1 provides an implementation for it. Method M3() is a concrete method of B, and it can also be replaced by a subclass.

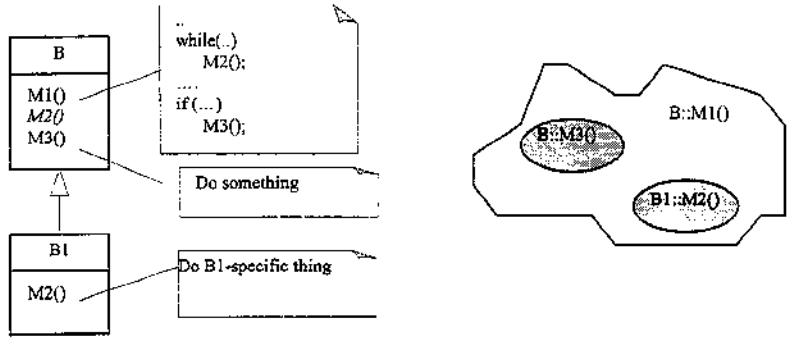


Figure 2: A template calling its hook methods.

[2] According to Pree, the term *framework example* is used to refer to those design patterns that describe reusable design structures, such as most of the design patterns from the pattern catalog of Gama *et al* [GHJV95] and some of the Coad's patterns [Coa92].

Pree defines seven metapatterns which describe the way the template method are implemented based on the hook methods and how the classes that implement them are related with each other. The class that contains the hook method(s) is considered as the *hook class* of the class that contains the corresponding template method(s), which is considered as the *template class*. The metapatterns describe how to compose template and hook classes and their corresponding objects. The seven metapatterns define the different combinations for these kinds of relationships, answering the following questions:

- Can an object of a template class refer to exactly one object of the corresponding hook class or to any number of objects of its hook class?
- Is the template class a descendant of the hook class? Are both classes unified?

The seven metapatterns are summarized bellow. For a detailed explanation of these metapatterns, we refer to [Pre95].

1. **Unification metapattern:** the template and hook methods are defined in the same classes, originating the unified template-hook class, which is represented as TH class in the Figure 3 (a).
2. **1:1 Connection metapattern:** the template and hook methods are defined in different class, and there is no inheritance relationship between these classes. An object of the template class refers exactly to one object of the hook class (this reference is represented as hRef in the Figure 3 (b)).
3. **1:N Connection metapattern:** this is similar to the 1:1 Connection. In this case, an object of the template class refers to any number of objects of the hook class (the reference is represented as hList in Figure 3 (c)).
4. **1:1 Recursive Connection metapattern:** the template and hook methods are defined in different classes, and an object of the template class refers to exactly one object of its hook class. The template class is a descendant of its hook class (Figure 3 (d)).
5. **1:N Recursive Connection metapattern:** this is similar to the 1:1 Recursive Connection. In this case, an object of the template class refers to any number of objects of the hook class (Figure 3 (e)).
6. **1:1 Recursive Unification metapattern:** the template and the hook methods are defined in the same class. An object of the unified class TH has a reference to one object of the same class TH (the reference is represented as thRef in Figure 3 (f)).

7. **1:N Recursive Unification metapattern:** this is similar to the 1:1 Recursive Unification. In this case, one object of TH refers to any number of objects of the same class TH (the reference is represented as thList in Figure 3 (g)).

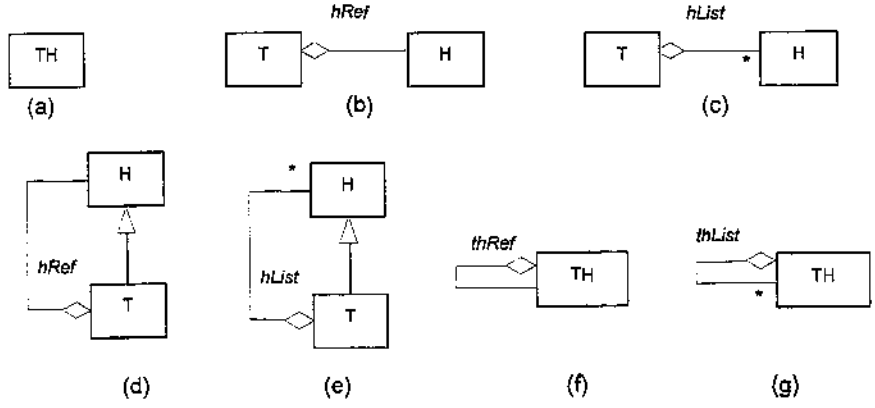


Figure 3: The seven Metapatterns

2.5 *Framework development*

The development of a framework is not a trivial task, since it should be designed to accomplish the requirement of a family of related applications. A well-designed framework should implement the common features of the framework's domain and provide adequate hot spots that satisfy its required variability [Pre95]. Primarily, domain-specific knowledge is required to identify the hot and frozen spots. After they have been identified, architectural styles can be applied to describe the framework's architecture at the high-level design, and design patterns and metapatterns can be applied to describe the class design of the hot spots.

We describe two approaches for the framework's development that is based on hot spots and patterns: the Pree's approach and the Schmid's approach.

Pree's approach

Wolfgang Pree proposes a hot-spot-driven approach to the framework development which is summarized in Figure 4. Once the desired hot spots are identified, the metapatterns are used to describe them identifying the template and hook methods, the classes that implement them and their relationships. The metapatterns describe the framework's adaptability independent of a specific domain.

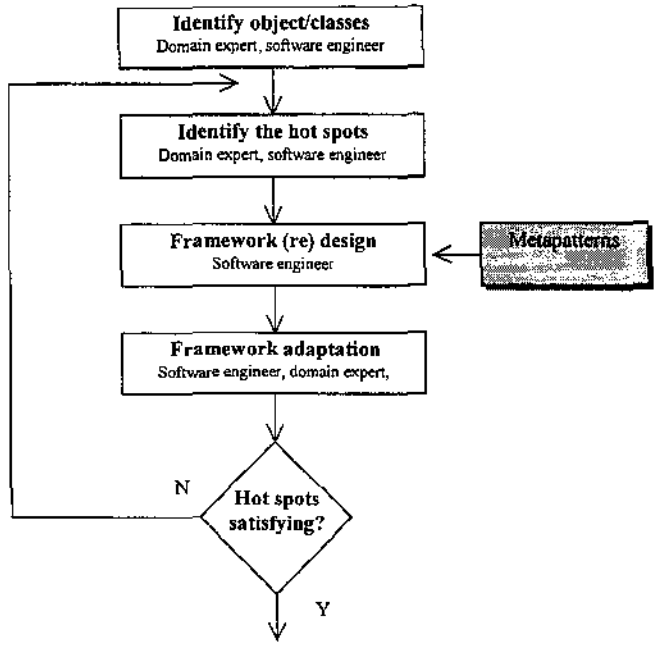


Figure 4: Hot-spot-driven approach using metapatterns to describe the adaptable parts

Schmid's approach

The Schmid's approach is also based on hot spots and frozen spots and is very similar to the Pree's approach. In addition, Schmid also proposes a method to identify the hot spots of a framework by applying a sequence of generalizing transformations in a specific model of an application from the domain [Sch97]. According to Schmid, the complexity of framework's design is reduced by separating clearly different issues: (1) the design of a class model for an application from the framework's domain; (2) the analysis and specification of the domain variability and adaptability; (3) the stepwise implementation of these variability by a sequence of generalizing transformations performed on the basic model. The generalizing transformations generate the hot spots, which are described using design patterns.

These two approaches are very similar, except by using different pattern approaches to describe the adaptable parts: Schmid applies the design patterns from the pattern's catalog [GHJV95] and Pree applies the metapatterns from [Pre95]. We have decided to use a combination of these two approaches, applying both design patterns and metapatterns for describing the framework's variability and adaptability. The design patterns capture the detailed structure and semantic of the adaptable design, the classes' collaborations and responsibilities. The metapatterns are more abstract and can be used to describe the points of adaptability of a specific design pattern in a meta-level. They describe with more precision how a hot spot can be

adapted by identifying the template and hook methods and the relationship between the template and hook classes of a design pattern.

Regarding the identification of the hot spots, our approach is more similar to the Schmid's approach. As a start point, we have the model of a specific application from the domain of train controllers, called Train Set System [Rub94, Qua97]. We have applied a sequence of generalizing transformations in this specific model based on the analysis of the domain, generating the hot spots that satisfy the adaptability of the train controller domain.

Our approach for the whole development of the framework also includes the design of the framework's architecture as the first step toward the framework design.

3 Basic Model of a Train Controller Application

This section describes the main features of the basic model of a specific train controller, called Train Set System [Rub94, Qua97], which has been used to generate the framework. This basic model presents the same restrictions that are also considered in the framework's design.

3.1 The Train Set System

The Train Set System is a digitally controlled model railway (Figure 5), which is divided into three parts: electronic digital units, railway layout and trains. The railway layout is mounted on three separated boards that can be independently controlled by separated controllers. Each board can be viewed as being composed of a set of switches, sensors and railway tracks, which link connectors and sensors (which we call stations, since they are the only source of information about the state of the system). Our case study is based on this railway model, but the development of the framework is limited to the software controller itself, since we do not have the original Marklin hardware (the digital units, the railway layout and the train engine) which was used to develop the Train Set System.

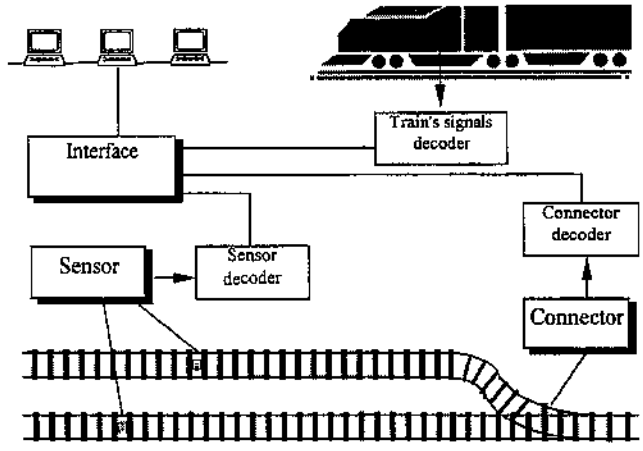


Figure 5: The Train Set system and the representation of the Marklin hardware

Sections are directed links of railway tracks between adjacent stations, and can contain a sequence of zero or more connectors. There are three different kinds of connectors: (i) a *crossing* is a static kind of connector which cannot be controlled; (ii) *end point* is a terminal connector; (iii) *switch* is a connector that has two controllable directions, straight and curved, and it can be of two kinds: *point* and *crossover*. Figure 6 shows the main kinds of connectors. There are also three different kinds of sections: (i) a *solid section* is a section that has next sections; (ii) a *partitioned section* does not have next sections and (iii) an *interconnected section* is located at the boundary of two boards, thus its next sections belong to another board. Figure 7 shows the layout of the three separated board controlled by three distributed computers, with examples of sections, stations, connectors and edges.

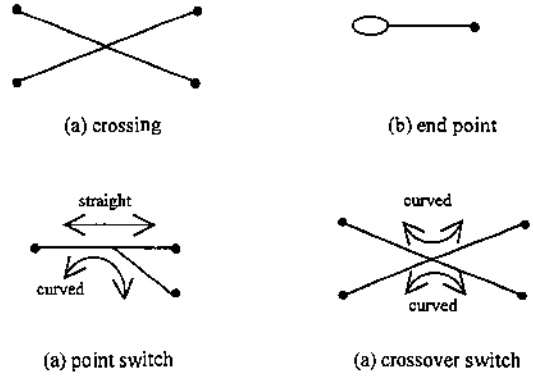


Figure 6: Kinds of connectors

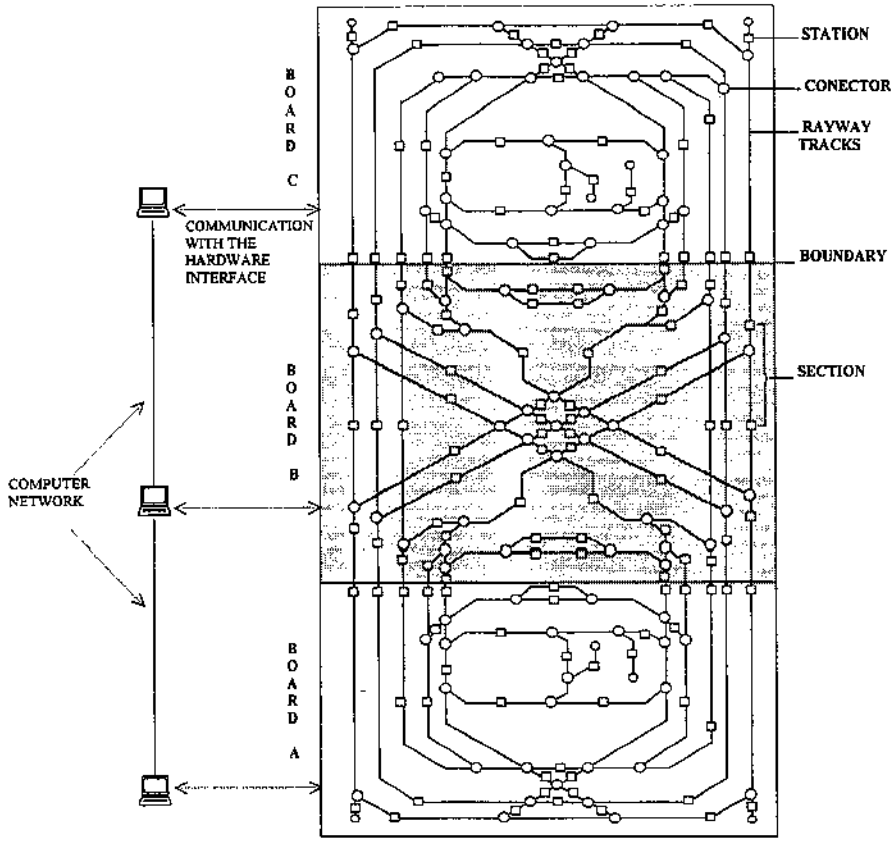


Figure 7: The railway layout composed by three separated boards

The trains aim to move randomly between stations. A train starts its movement in an initial section of the board and continues its trip accordingly to the availability of the next sections. If a next section is either reserved for another train or faulty and consequently cannot be used, the train tries to localize another one. Despite the presence of faulty switches and sensors, the trains should move around the railway without crashing, but if necessary stopping and reversing. Since the railway layout is divided up in three boards, the design should take into account both the layout distribution and the train crossings between neighbor boards.

The relevant restrictions of the train controller application can be summarized as following:

- The main goal is to guarantee no train collision, i.e., safety;
- Switches and sensors are unreliable devices and can suffer environmental faults. However, considering two consecutive sensors it is assumed that only one sensor can fail.
- Derailment of train is not considered.
- Routing of the train is ignored, i.e., the trains move randomly between stations.

- Train can stop within one section that means that it travels slowly enough to stop immediately when requested.
- We assume that the train size is smaller than the smallest section and can be completely contained within a section.

The most important non-functional requirement of train controller applications is dependability, since it is a critical system: a failure in the controller system can lead to the collision of trains. Fault tolerance techniques can be used to achieve the dependability requirement, avoiding faults from causing failures in the system. We have concentrated our attention on environmental faults that entities with which the system interacts can suffer. The main sources of environmental faults in the train controller system are switches and sensors. We assume that all the other elements or devices of the train set are reliable. Another non-functional requirement is distribution: the three boards can be controlled by three independent computers, which are connected by a network.

3.2 *Basic class diagram*

The previous works about the Train Set System [Rub94, Qua97] have an extended explanation of the process of defining the basic object model of the application based on the requirements. In this paper, we present only the most important class diagrams of the Train Set System that are important for understanding the framework's design. Figure 8 shows the most general class diagram of the system (using the UML notation).

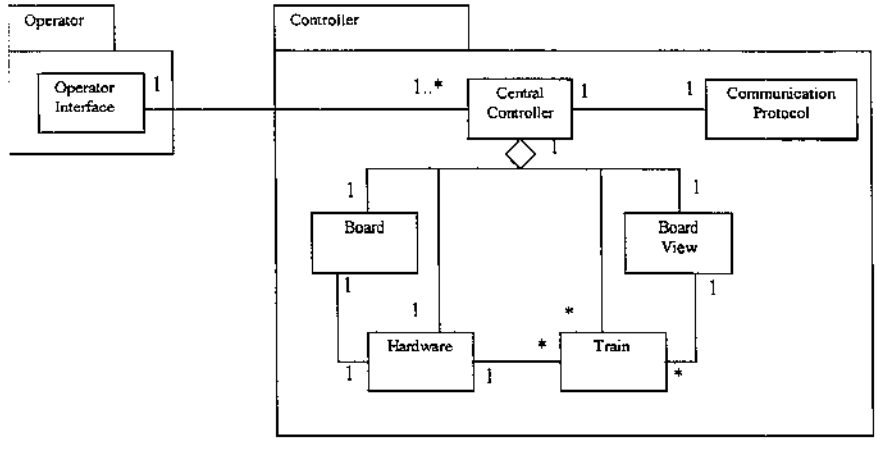


Figure 8: Basic class diagram of the Train Set System

The system is composed by two main packages: the Operator package and the Controller package. The Operator package defines the OperatorInterface class which implements the interface

with the operator of the system who is responsible for creating trains, specifying how trains should be first located, where they should go, what precautions should be taken against collisions, and when to start and stop the system. The Controller package implements the main functionality of the system. It defines the CentralController class, which is the core of the application. The CentralController is broken down into many control objects, such as train controllers (one for each train) and board controller. More specifically, the CentralController is an aggregation of one instance of the Board class, many instances of the Train class, one instance of the Hardware class and one instance of the BoardView class. The CentralController is also associated with one instance of the CommunicationProtocol class, which implements the communication protocol between two distributed controllers. The CentralController and the OperatorInterface are associated with each other, and the OperatorInterface should pass messages to the CentralController initializing the system, and the CentralController should pass messages to the OperatorInterface about relevant system information such as the position of the trains and about faulty states of the system. The other classes of the Controller package are:

- *The Train class,* which is an important control object of the system, representing trains moving around the board. A train requests services such as lockSection() and releaseSection() to the Controller, updateTrainPosition() to the BoardView, and setSpeed() to the MarklinInterface. It implements methods such as start(), move(), reverse() and stop().
- *The Board class,* which represents the board components. The Board class is a composition of many instances of Sections. The Section class is a composition of two instances of Stations (head and tail) and zero or more instances of Connectors. The Connector and Station classes are associated with the Edge class (Figure 9).

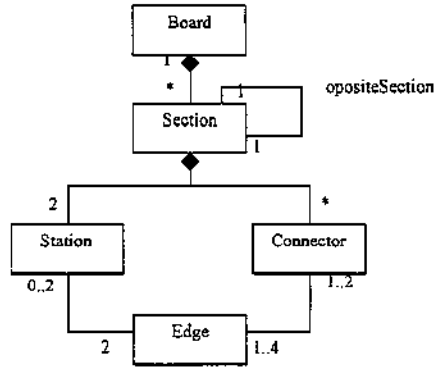


Figure 9: Class diagram of the Board component

- *The BoardView class*, which represents the railway's layout and reflects the state of the system, such as the position of trains, free and locked sections, and also, exceptional states about faulty components. The view can be updated by the Train instances and by the Controller.
- *The MarklinInterface class*, which implements methods of interaction with the railway hardware, including the board devices (switches and sensors) and the train engine.
- *The CommunicationProtocol class*, which implements the details of basic functionality for the communication between two distributed controllers. It encapsulates details about inter-process communications between process that are in different address space.

3.3 Fault tolerance

As stated before, the Train Set System should tolerate environmental faults of switches and sensors. The implementation of fault tolerance mechanisms in the Train Set system involves two issues: (i) error detection and recovery and (ii) fault treatment.

3.3.1 Error detection and recovery

The train should be capable of detecting an error in its current position (caused either by a faulty switch or a sensor that was triggered erroneously) and of recovering its position avoiding train collisions. The concept of control zone is very important for avoiding train collisions. The control zone is the front region acquired by a train, i. e., all sections locked ahead of the current position of the train (the next sections). Each train has a control zone and is responsible for setting its route within its control one. The control zone is constructed with one or more levels of next sections. The first level holds information of the next sections of the current section of the train. The second level holds information of the next sections of each section of the first level, and so on. Figure 10 shows an example of a two-level control zone.

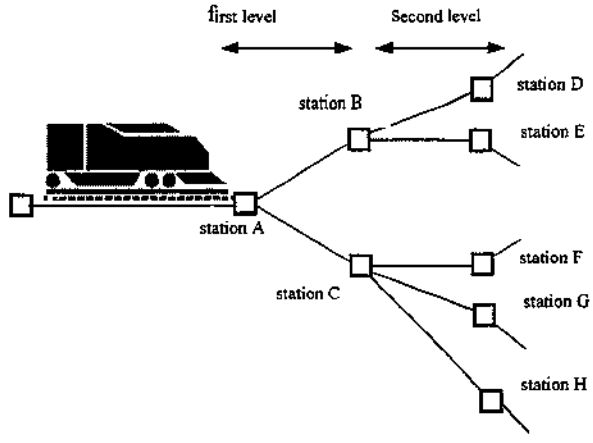


Figure 10: Example of a control zone of one and two levels.

If we assume that all devices are reliable, it is enough to lock only one next section of the current section in order to avoid train collision. However, if we assume that switches are unreliable devices (and sensors are reliable) the train needs to acquire a control zone with one level (all the next sections of the current section), while if we assume that both switches and sensors are unreliable, the train needs to acquire a two-level control zone.

The train is capable of detecting an error in its position based on an exception handling mechanism. The train knows what is the next sensor to be triggered. When an unexpected sensor is triggered outside its control zone, the train signals an exception to the controller. After detecting the error, it is necessary to remove the errors from the system state, by means of error recovery techniques. The Train Set System implements forward error recovery: when a train detects an error in its position, it tries to set a new route within its control zone in order to recover from the error, or if it is not possible, the train should stop and wait or reverse.

The design solution defines a hierarchy of Train classes to implement the tolerance of each kind of possible fault (Figure 11). The Train class is the base of the hierarchy and does not tolerate any kind of fault, i. e., it assumes that the switches and sensors are reliable, and the control zone is only the next section of the current section. The FTConTrain class implements tolerance of switches, defining a control zone of two levels. The RobustTrain class implements tolerance of both switches and sensors, defining a control zone of three levels. The move() method is overridden to implement the different control zones.

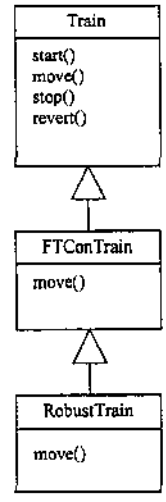


Figure 11: The Train class hierarchy to implement different error detection and recovery

3.3.2 Fault treatment

After performing the error treatment, is still necessary to treat the fault to prevent it from continuing to damage the system state. For that, the system should be reconfigured properly. The reconfiguration consists in changing the behavior of the faulty components, so that their service implementations reflect the faulty (abnormal) state of the component. This reconfiguration should be performed dynamically, since the system cannot stop. The reconfiguration strategy consists in encapsulating the abnormal behavior phases of faulty entities as objects, and developing stand-by variants of this abnormal behavior phases to replace the behavior implementation of faulty components.

This solution is implemented as the State design pattern [GHJV95]. We define a state class hierarchy parallel to the component class. The normal and abnormal behaviors of the component are encapsulated by the state concrete classes. The component delegates the execution of its services to its current object. The component changes its behavior by changing its current state object. Figure 12 shows the design of the Switch class and its correspondent SwitchState hierarchy. The section class is designed similarly.

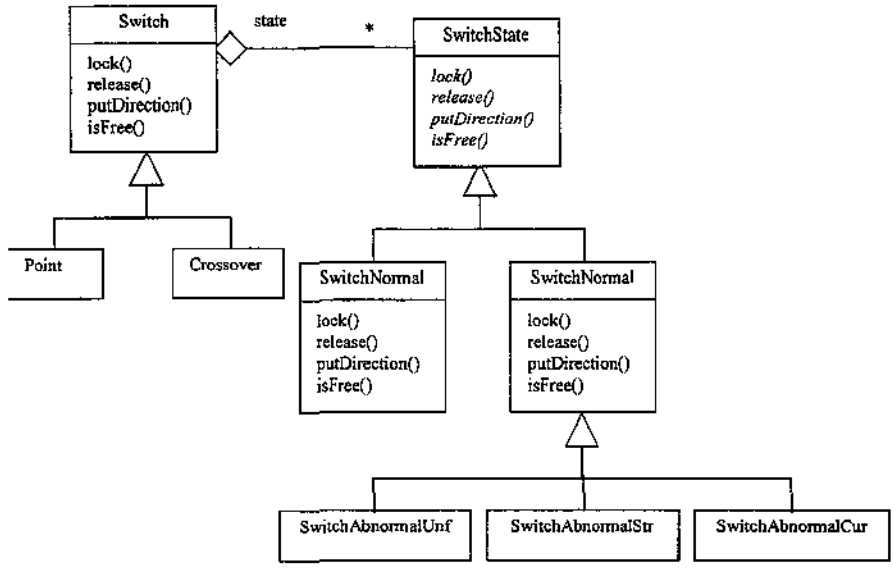


Figure 12: The design of the Switch component using the State design pattern

4 The Framework Design

As stated before, a framework for an application's family defines the fixed parts of the domain, i. e. the frozen spots, and the parts that should be kept flexible to fill the specific applications' features, i.e. the hot spots. Based on the specific model of the Train Set System and on the domain analysis, we have identified the frozen and hot spots, looking for the commonalities and variabilities of this domain.

4.1 The frozen spots and the framework architecture

Some of the frozen spots of the train controller domain are:

- The central controller
- The control of the set of trains or other mobile objects: the framework implements the control of the trains or other mobile-object's movement around the board, although the kinds of mobile objects can be different for each application.
- The control of sensors that detect the position of trains in the board.
- The control of actuators, i. e., the switches or other kind of actuators that perform alterations in the environment.
- The state machine implementation. The framework implements the control aspects related to the state machine that is executed in the State design pattern. (execution of the state

transitions and creation of state objects), although the specific configuration of this state machine can be different for each application.

The framework architecture descriptions define the main components of the framework, their collaborations and restrictions. The framework should implement the fault tolerance techniques in its fixed parts, what increases its complexity due to the introduction of component redundancy and exception handling mechanism, as discussed in the section 3.3. We have identified the architectural styles that provide better solutions for the design of redundancy and exception handling in the fixed part of the framework, keeping the design complexity under control. The next sections briefly describe the architectural styles and the resulting architecture descriptions of the framework.

4.1.1 Architecture description using the Idealized Fault-Tolerant Component style

The *Idealized Fault-Tolerant Component* style is based on a well-known model for constructing fault-tolerant systems: the idealized fault-tolerant component model proposed by Lee and Anderson [LA90]. According to the model, a system can be viewed as a set of components interacting under the control of a design (that is itself a component of the system). The system model is recursive in the sense that each component can itself be considered as a system on its right, and thus can have a recursive structure composition which identifies further sub-components. Moreover, these components receive requests for service and produce responses. If a component cannot satisfy a request for service, then it will return an exception. At each level of the system, an idealized fault-tolerant component will either deal with exceptional responses raised by components at a lower level (a supplier component) or else propagate the exception to a higher level of the system (a client component).

Figure 13 illustrates the components, connectors and the design rules of this style. The components of the style are the supplier and client components, which are divided in two parts: (1) the normal part that implements the normal service and returns normal responses and (2) the abnormal part that implements the measures for handling exceptions returned by a supplier component. The connectors of the style are the service requests and service responses, which are used by the components to communicate with each other. In the *Idealized Fault-Tolerant Component* style, interface exceptions are signaled in the normal part of the component, while

failure and interface exceptions from supplier components invoke the exception handling part of the client component. If these exceptions are handled successfully (that is, the component was able to mask the exception), the component can return to providing normal services. However, if the component does not succeed in dealing with such exceptions, it should signal a failure exception to a higher level of the system.

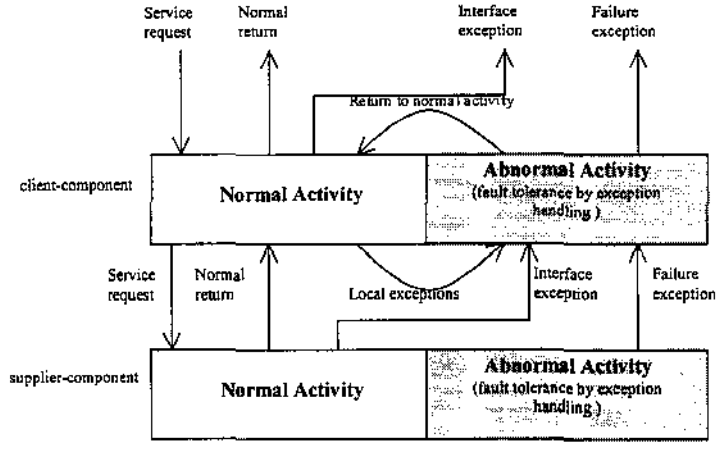


Figure 13: The Idealized Fault-Tolerant Component style

In order to describe the framework's architecture using this style, we identify the fault-tolerant components that interact to perform the main functionality of the application. At a higher level of abstraction, we identify three main fault-tolerant components: Train, Controller and Board (Figure 14). These components represent packages of an object-oriented system, which encapsulate the details of their classes. The packages provide the public services that are in turn provided by their constituent classes.

The Train component is responsible for the movement of trains, implementing services such as move(), stop(), revert(), etc. The Controller component is the central part of the system, and it is the intermediate component between the Train and the Board. The Controller provides important services to the trains so that they move around the boards without crashing, such as lock_section(), release_section(), etc. The Board component defines the detailed representation of the board's layout, and is composed by a set of connectors, sensors and sections. It is responsible for creating, initializing and managing these components.

As stated before, the goal of the system is to control the movement of trains so that no train collision occurs. For that, a Train requests services to the Controller, such as lock_section(), release_section() and occupy_section(). If the service is executed normally, the Train receives a

normal response and continues moving toward the next section. If some fault occurs in a supplier-component, the service request cannot be performed normally and an exception is signaled and propagated to the Train. For instance, the Board can return an exception as a response of the operation lock_section() due to some faulty sub-component (e.g. a switch that belongs to the required section is faulty and the section cannot be used by a Train). The exception is propagated through the components that are between the lower-level component, i. e. the switch device, and the higher-level component, i.e. the Train. The Train is the component responsible for handling the exception, using a forward error recovery mechanism: if a requested section is abnormal, the train attempts to lock another one in order to recovery from the error.

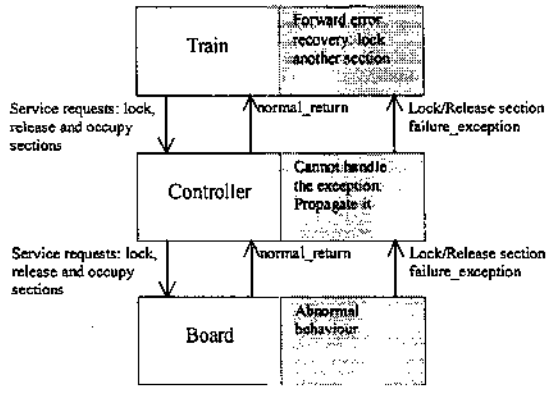


Figure 14: The architecture of the dependable framework for train controllers using the Idealized Fault-Tolerant Component style

4.1.2 Architecture description using the Meta-Level style

Meta-level architectures are based on the computational reflection concepts. Computational reflection is a technique that allows a system to maintain information about itself (meta-information) and use this information to change its behavior (adapt) [Mae87]. This means that a component can get information about the internal properties of another component and based on that information it can dynamically interfere on its current computations.

The main benefit of object-oriented meta-level architectures is the modularization of the system in at least two levels (or layers): the meta-level and the base level. The meta-level encompasses the objects that deal with the processing of self-representation and management of an application, and the base level encompasses the objects responsible for implementing the functionality of the application. Figure 15 illustrates this style.

For the meta-level to be able to reflect on base-level objects, it must be given information regarding the internal structure of base-level objects (structural meta-information). The representation, in form of objects, of abstract language concepts, such as classes and methods, is called *reification*. Moreover, interaction between objects may also be materialized as objects, so that meta-level objects can inspect and possibly alter them. This is achieved by *intercepting* base-level operations such as method invocations, creating objects that represent them, and transferring control to the meta-level. After transferring the control to the meta-level, the meta-objects can inspect the reified information and can also modify structural and/or behavioral aspects of the base-level object. This process is also called *reflecting* the changes back into the base-level object [Mae87, Fer89].

The connector of this style is the meta-object protocol (MOP), which establishes the relationship among the base-level and meta-level objects. The MOP provides an interface to the programming language implementation in order to reveal to the program information normally hidden by the compiler and/or run-time environment [Mae87]. The MOP's kernel is responsible for implementing the interception, reification and reflection mechanisms described above.

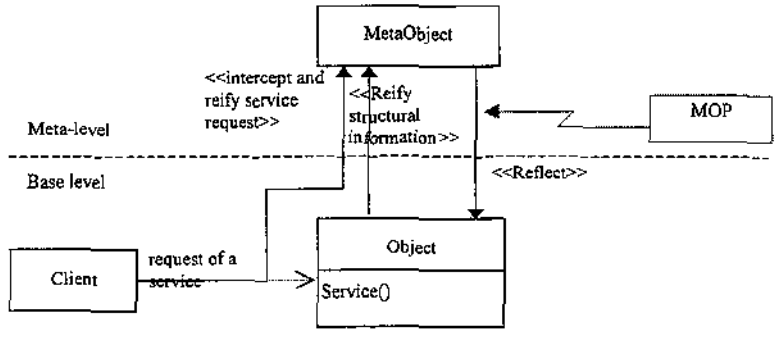


Figure 15: The Meta-Level architectural style

Regarding the development of fault-tolerant systems, the *Meta-Level* style can be used to separate the management activities related to the control of component redundancy that is used to implement fault tolerance. In the previous section 3.3.2, we have presented the design for implementing environmental fault tolerance by means of state classes that implement the normal and abnormal behavior of a faulty component. These classes represent the redundant components since they are not necessary if we consider that the system is free from faults. In fact, there is a state machine execution behind this design that controls the execution of the state transitions.

Following the *Meta-Level* style, we can define the control aspects related to the state machine execution at the meta-level, separating them from the functional aspects implemented by the component's classes. Figure 16 shows the structural organization of the framework using the *Meta-Level* style. The fault-tolerant components (FTComponents) define redundant components needed to implement environmental fault-tolerance, and the MetaStateMachine component implements the control aspects related to the execution of the state machine.

We have used the Guarana's MOP [Oli98] to implement the MetaStateMachine. The Guarana's kernel implements the interception, reification and reflection mechanisms. The Guarana reflective architecture also defines some basic classes that can be derived to implement the meta-objects behavior.

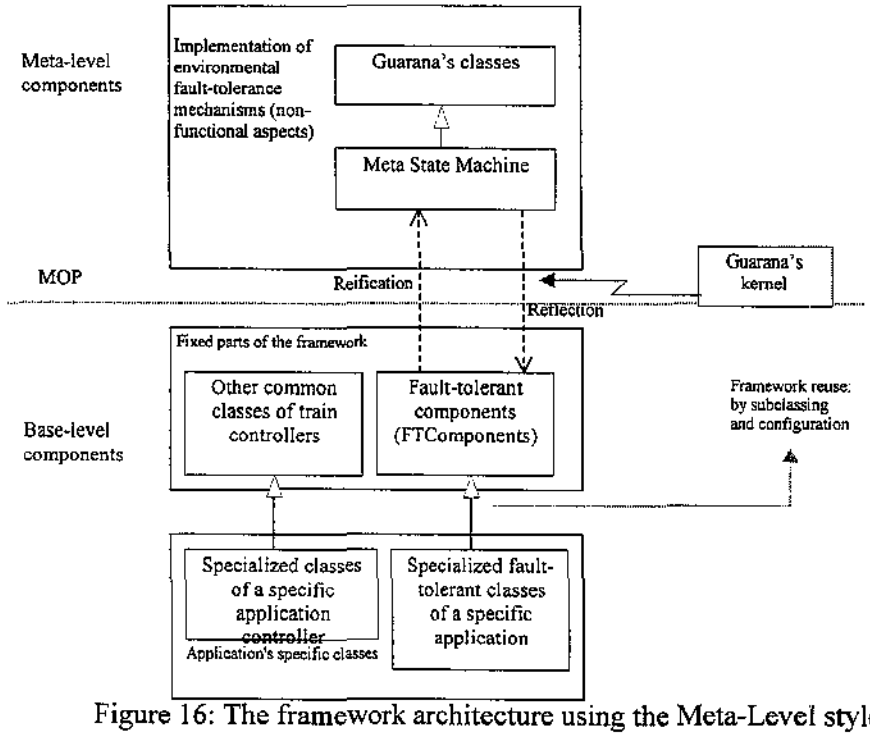


Figure 16: The framework architecture using the Meta-Level style.

4.2 The hot spots

The main adaptable parts of the framework are:

- *The composition of the board:* different applications have different compositions of the board, depending on the railway layout. The framework should support the definition of any board's format.
- *The board view* is also an adaptable part since it reflects the board composition. The framework does not define any implementation for the board view; it implements only an

abstract class that defines the interface (abstract public methods) that the board view module of a specific application should implement.

- *Different kinds of mobile objects*: the specific application can extend the Train class to implement different error treatment for the different kind of faults, and can also define different kinds of mobile objects with similar functionality, such as wagonettes.
- *Fault tolerance*: each specific application can redefine the possible states of the components, changing the state machine configuration and the State class hierarchies of the fault-tolerant components.
- *The communication protocol:* each specific application can redefine or replace the communication protocol between the distributed controllers.

In the next sections, we describe the hot spots of the framework identifying the following aspects:

(i) the adaptability;

(ii) the problems related to the required adaptability in the basic model;

(iii) the requirements that the solution should implement;

(iv) the design pattern/metapattern that describes the adaptability. Wherever is identified a design pattern and metapattern, we use a combination of both. Otherwise, if no semantic-specific design pattern is identified, we use only metapatterns that are more abstract.

4.2.1 Hot spot for the board's composition

Adaptabilty

The composition of the board should be kept flexible, since it depends on the railway layout. Different applications can have different board compositions.

Problem

The basic model of the Train Set System defines a specific board composition that implements its railway layout. The railway is composed by three separated boards, each board is composed by sections and each section is composed by specific kinds of sensors and connectors. We can classify the board's components as primitive and composed. A section is a kind of composed component, and connectors and sensors are primitive components. The layout of these

components and the composition of the sections are specific for the Train Set system, and the creation of these objects depends on the specific railway layout.

Requirement

The solution should allow the composition of any kind of board, with different layout for its components. The composition of the composed objects should be kept flexible so that a different composed object can be created by a different combination of primitive objects. Furthermore, the solution should allow the creation of specialized types of primitive objects (for instance, specific kind of connectors, stations, and edges) and composed objects (for instance, specific kind of sections or other kind of composed object).

Solution

We apply the Composite design pattern [GHJV95] to implement flexible composition of objects (Figure 17). Following the Composite pattern, we define an abstract base class called BoardItem, which is the base of the hierarchy, and a recursive hierarchy of primitive and composed classes. The Block class implements the recursive composed class, implementing operations to add and get objects of BoardItem class, and other operations that are executed uniformly in all its objects, such as lock(), release(), etc. The Section class is a specific kind of Block composed by two stations (head and tail stations) and any number of connectors. A specific application can define new types of board items by inheriting the primitive classes. New kind of blocks can be created using different composition of these specialized objects.

We apply the 1:N Recursive Connection metapattern to describe the adaptability for the composition of the board components. The Block class is a template class and implements the template methods lock(), release(), isFree(), which call the hook methods (with the same name) of its BoardItem objects.

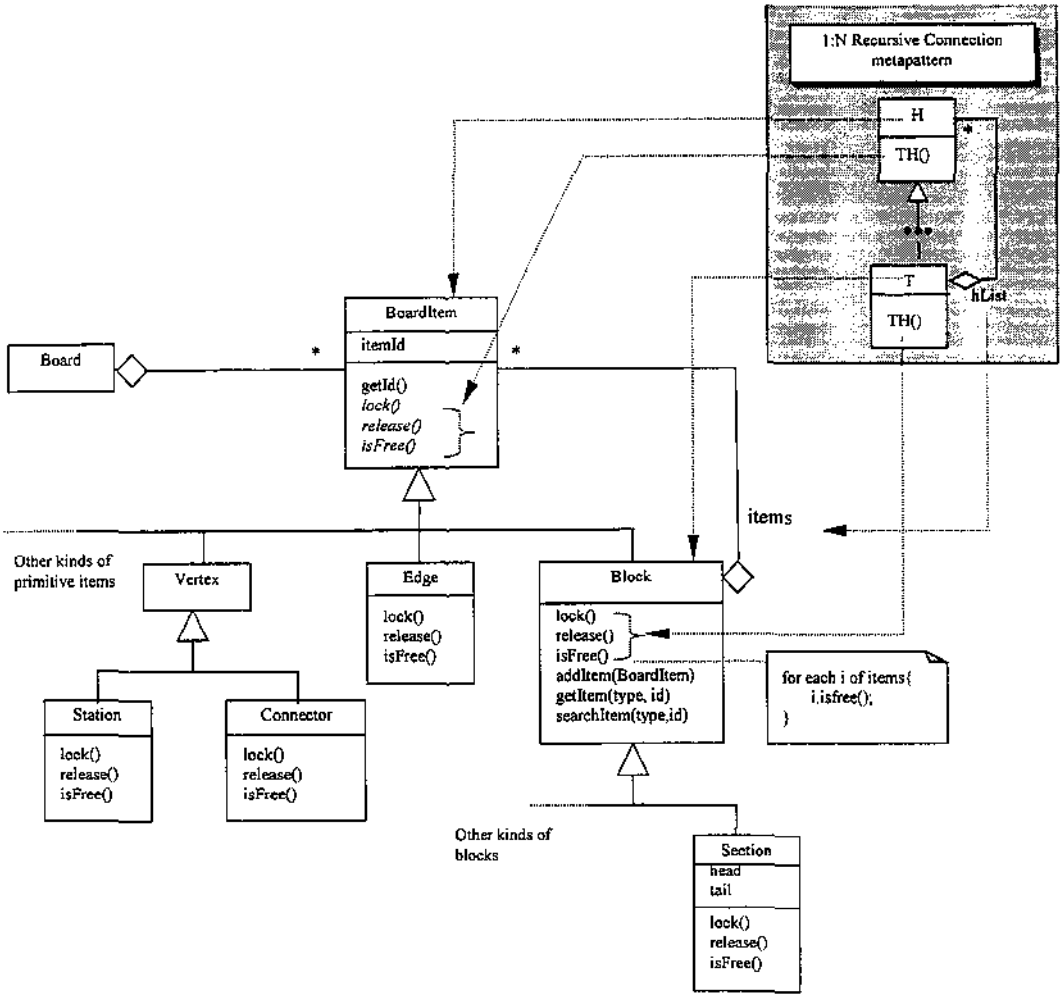


Figure 17: Flexible structure for the composition of the Board's components.

The Board class encapsulates the information about all board's components, maintaining the list of all connectors, stations, blocks, etc. It is responsible for creating and initializing these objects accordingly to a specific layout configuration, and also for providing the access of them for other objects, such as the Controller and the Train. The Board class is implemented as the Manager pattern [SB98], which treats the collection of objects as a whole, and allows the access for each object independently of its specific type (Figure 18). For instance, the Controller object (which implements the role of a client) retrieves Section objects from the Board and call services in these objects. The Board class also implements an Abstract Factory pattern [GHJV95], allowing the flexible creation of the collections of objects that compose the board, so that the specific application can redefine the types of the primitive and composed components.

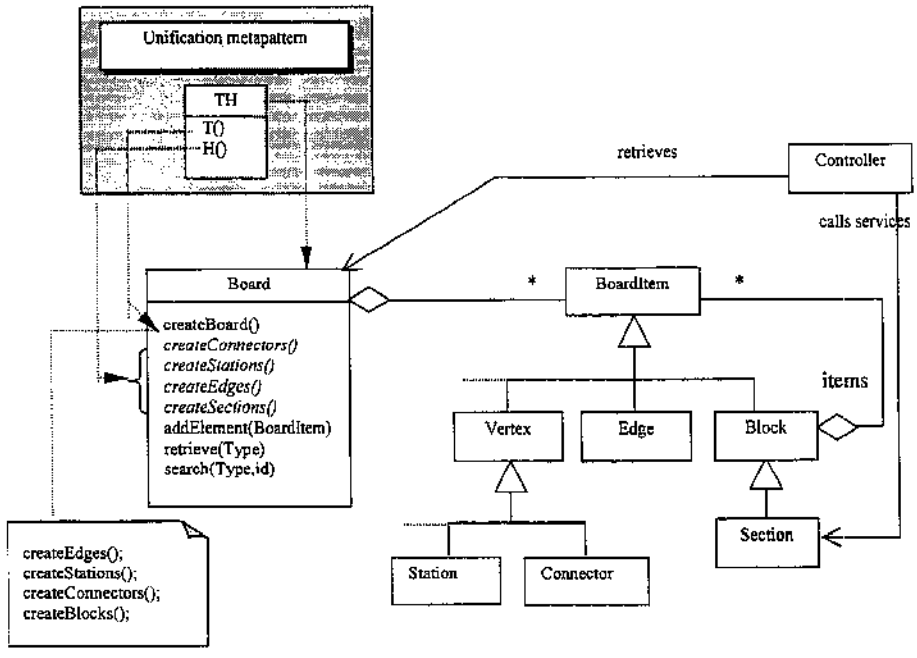


Figure 18: Flexible structure for the creation and management of the board's components

The Unification Metapattern is applied to describe the Abstract Factory pattern. The Board class implements the template method createBoard() that calls the abstract hook methods *createConnectors()*, *createStations()*, *createEdges()* and *createBlocks()*. A specific application should provide an implementation for these methods to create the specific kind of board's objects accordingly to the specific board's layout.

4.2.2 Hot spot for the board's view

Adaptability

The board's view should be flexible to represent any railway's layout and composition of board's components. The view update by the controller and trains should be independent of the specific implementation of the view.

Problem

The board's view represents the railway's layout and it reflects relevant state information about sensors, switches settings, positions of the trains in the board and reserved/free sections. The board view implementation of the Train Set System reflects the specific composition of the board accordingly to the specific railway layout. The trains and controller depend on this specific implementation to update the view.

Requirement

The controller and trains should be capable of updating the view about relevant state information of the system independently of a specific implementation of the view.

Solution

The framework does not define a specific implementation for the board view since it should reflect a specific railway layout. The framework defines only the interface for the view update, so that the controller and trains can be unaware about the specific implementation of the board's view and how the separated views are updated consistently. The controller and train update the view through predefined messages that represent the possible changes in the system's state.

We use the Observer design pattern [GHJV95] to implement the update of the board's view independently of a specific view implementation (Figure 19). The BoardView class is an abstract class that defines the abstract method update(), which receives a message that carries the information about the changed state. Each concrete implementation of the BoardView class overrides this method, providing the specific actions to update the interface reflecting the change of the system's state. The Observable class implements the methods attach() and detach(), which are responsible for initializing the Observable object with one or more instances of a concrete implementation of the BoardView class. It also implements the notify() method, which calls the method update() on the BoardView object. The Train and Controller classes inherit from the Observable class, so that they can attach/detach BoardView objects, and notify them about relevant state changes. For instance, the Train object updates the view with information about free/locked sections and its position, and the Controller updates the view about faulty system's states.

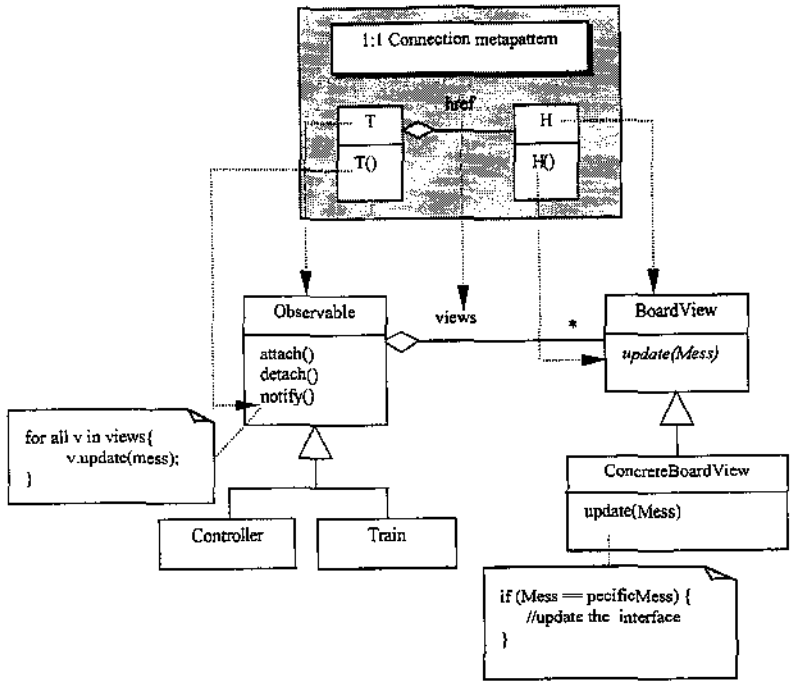


Figure 19: Flexible implementation of the board's view.

A concrete BoardView class can implement either the representation of the entire board or the representation of each part of the board. In the last case, each node has its correspondent board view and the Controller and Train instances are initialized with one instance of the concrete view, and update only this view (the attribute views in the Observable class is a list with only one object). If the concrete BoardView class implements the entire board, each node has a BoardView object that shows the state of the entire board. In order to guarantee the consistence among the views, the Controller and the Train instances should update not only their own view, but also the other available views. In this case, the Train and Controller instances are initialized with a list of the available views and call the update() method in these BoardView objects. The pattern treats these two cases uniformly: the update() method is implemented by the Observable class in the same way in the two cases, and the Controller and the Train instance do not care about the kind of the BoardView object and how it is updated.

4.2.3 Hot spot for the creation of the mobile objects

Adaptability

Each specific application can control different kinds of mobile objects. For instance, different applications contain specialized types of trains or different mobile objects with similar functionality, such as wagonettes.

Problem

The Train Set System considers one type of mobile objects, that is train. It defines a hierarchy of Train classes that implements the different kind of error treatment (error detection and error recovery). The basic class is Train, which does not tolerate any kind of fault. There are two specialized kind of trains: (1) the FTConTrain, which implements the error treatment for connector faults and (2) the RobustTrain, which redefines the FTConTrain to implement the error treatment from connector and sensor faults. These types of trains are specific for the Train Set System, and the creation of the objects is not transparent for the application.

Requirement

Each specific application should be able to redefine new kinds of trains, which implement different kind of error treatment, and also treatment for other kinds of faults. The framework should be independent from the creation of the specific type of train or other types of mobile objects with similar functionality.

Solution

We use the Prototype design pattern [GHJV95] to implement the flexible creation of different types of mobile objects (Figure 20). The framework defines the base class MobileObject, which defines the abstract method clone(). The concrete subclasses of MobileObject should provide the implementation for this method, returning a copy of itself. Then, an instance of MobileObject is a prototypical instance that is used by the application to clone specific types of mobile objects.

The framework also defines a MobileObjectManager class, which is responsible for creating new instances of a specific MobileObject using the prototype object. The MobileObjectManager maintains an abstract reference for a MobileObject (moPrototype), and it is parameterized by an instance of a concrete subclass of MobileObject. The createMobileObject() method returns a new instance of the concrete MobileObject by calling the clone() method of moPrototype.

The Controller object is responsible for creating and inserting trains, and it should be independent of the type of train that should be instantiated. For that, the Controller is initialized with an appropriated MobileObjectManager object, which creates the specific kind of MobileObject.

The Controller inserts MobileObject by calling the method CreateMobileObject() of its MobileObjectManager.

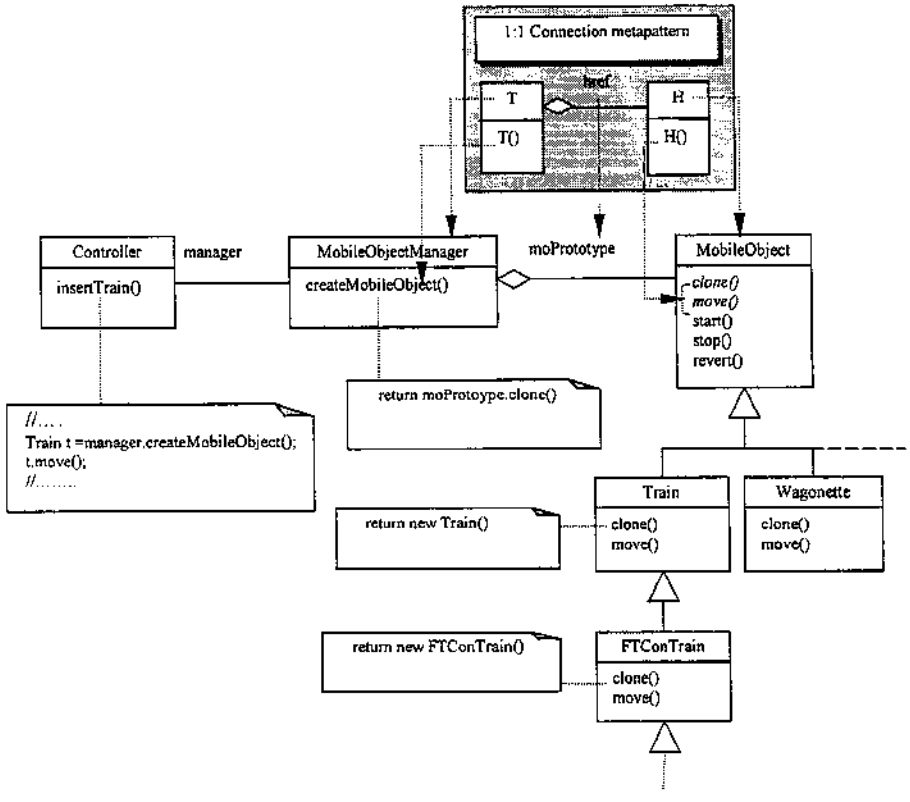


Figure 20: Hot spot for the creation of mobile objects

4.2.4 Hot spot for the fault tolerance

Adaptability

In order to implement environmental fault tolerance, we consider two basic states for the fault-tolerant components: normal and abnormal. The behavior in each state is implemented by a State class. Each specific application can redefine the behavior of each state and/or define different normal and abnormal states, extending the State class hierarchy and changing the state machine execution accordingly with a specific state diagram defined for the fault tolerant component.

Problem

The Train Set System design uses the State design pattern to implement a hierarchy of State classes, which implement the normal and abnormal behavior of the fault-tolerant components (switches and sections). The component holds the reference for its current State object and uses the delegation mechanism to delegate the state-dependent service to its current State object. The

component changes their states by changing the reference for the current State object. The control aspects related to creation of the State objects and the execution of the state transitions are implemented by either the fault-tolerant component or the State classes. In this solution, the control aspects related to the state machine execution and the functional aspects of the component (implementation of its services) are implemented together, what makes it difficult to extend the State class hierarchy and the fault tolerant component hierarchy independently.

Requirement

A specific application should be able to redefine the state diagram of the fault tolerant component, adding or removing states and transitions, and consequently, extending or redefining the State class hierarchy. These extensions should be independent of the extensions of the component's functionalities. Then, the solution should separate the control aspects related to the state machine execution from the functional aspects of the component.

Solution

We use the Reflective State pattern [FR98a, FR98c] that is a refinement of the State design pattern based on the Meta-Level architectural style (also documented as an arquitectural pattern named "Reflection" [BMRS+96]). It applies the Meta-Level architectural style to separate the application in two levels: (i) the base-level, where resides the objects responsible for implementing the functional activities of the application and (ii) the meta-level, where resides the meta-objects responsible for implementing the management and control activities. The Reflective State pattern implements the control of the state machine execution in the meta-level, separating it from the functional services implemented by the fault-tolerant component and the State classes.

Figure 21 shows the design of the switch component using the Reflective State pattern. At the base-level, we define the Switch class hierarchy and its correspondent SwitchState class hierarchy. These classes implement the normal services and the state-dependent services of the switch respectively, without implementing the control aspects related to the creation of the state objects and execution of the state transition. At the meta-level, we define the meta-objects that correspond to the state machine elements: the FSMState class, the FSMTransition class and FSMController. The FSMController instance represents the state machine controller, and the instances of the FSMState and FSMTransition represent the states and the state transitions of the state machine, respectively. Using the interception and reification mechanisms provided by the

reflective architecture, the FSMController meta-object intercepts and handles the operations sent to the Switch object. The FSMController delegates the handling of the operation to the current FSMState meta-object and to the FSMTransition meta-objects. These meta-objects perform the extra-computation related to the state machine execution. The Switch and SwitchState classes are completely independent, and each hierarchy can be extended without affecting the other.

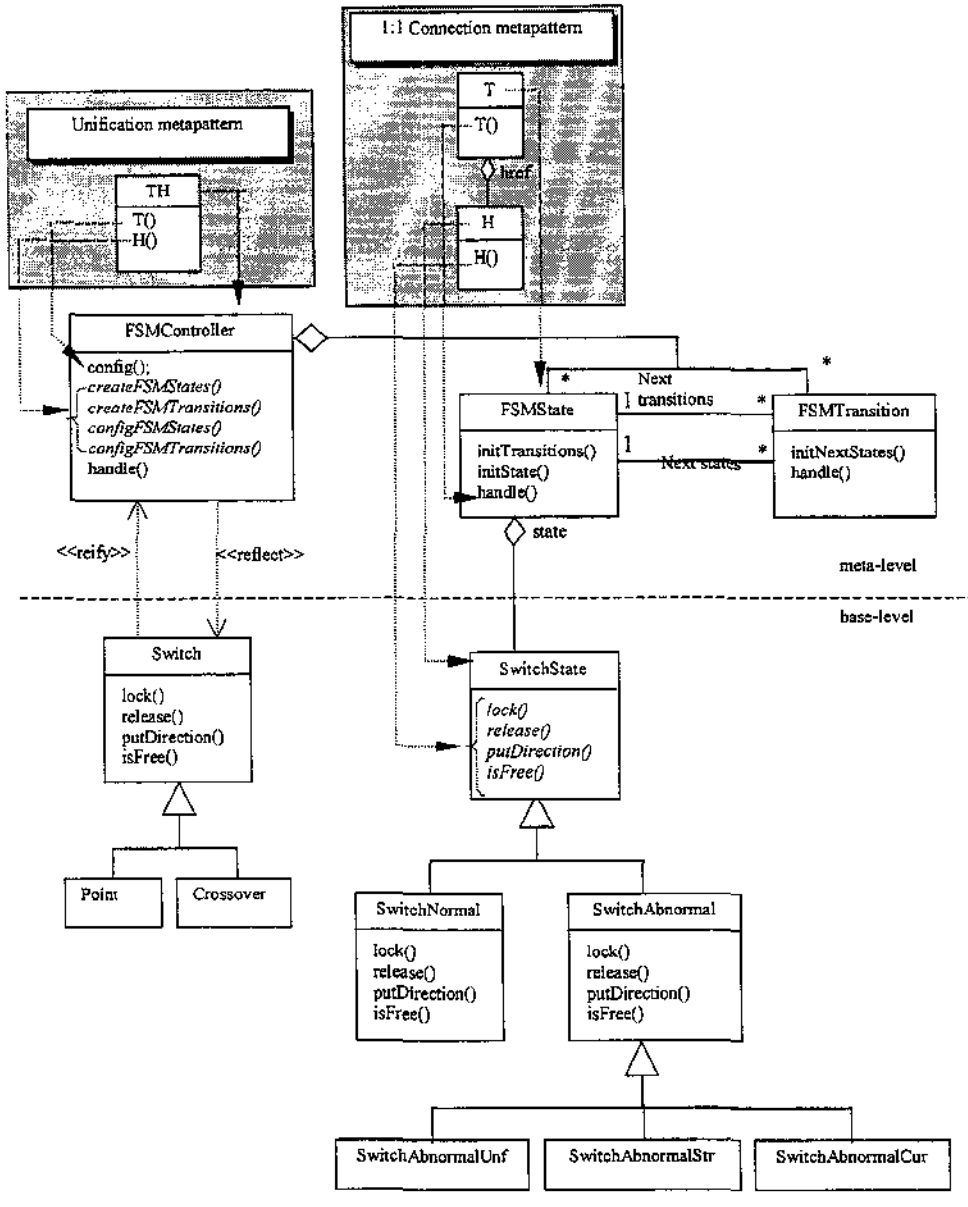


Figure 21: Design of the Switch component using the Reflective State Pattern

This solution represents two hot spots: (i) the adaptabilty for the redefinition of the state machine at the meta-level (creating new states and/or state transitions) and (ii) the redefinition of the state-dependent methods in each concrete State class at the base-level. Two metapatterns are

applied to describe these hot spots. The Unification metapattern describes how the FSMController can be redefined to implement a specific state machine. The template method config() calls the hook abstract methods createStates(), createTransitions(), configStates(), configTransitions() that should be redefined in a concrete subclass of FSMController to configure a specific state machine, creating and configuring the various FSMStates and FSMTransitions that represent the state-machine's elements. The 1:1 Connection metapattern documents the adaptability to redefine the state-dependent methods implemented by the State subclasses. The FSMState class is a template class that implements the template method handle(). This method uses some reflection mechanisms to execute an indirect delegation of the state-dependent service to a SwitchState object (more details of this mechanism can be obtained in [FR98a]). The SwitchState class is a hook class that implements the hook abstract methods lock(), release(), etc. A specific application can extend the SwitchState hierarchy by redefining new SwitchState subclasses, overriding the hook methods.

4.2.5 Hot spot for the communication protocol

Adaptability

Each specific application can redefine or change the communication protocol used by the distributed controllers to communicate with each other.

Problem

The Train Set System is a distributed application and thus, it implements a specific distribution mechanism to perform the communication between the distributed controllers over the network, using a specific communication protocol. Furthermore, the controllers of the application play the role of a client and a server at the same time. It complicates the design, since each object should implement the two sides of the communication protocol: the client and the server side.

Requirement

The solution should implement a transparent interface for the communication between the distributed objects, so that they are unaware about the specific protocol used to implement the communication over the network. The application's objects should be unaware about the location of the objects with each they interact, i. e., if the objects are local or remote objects (this property is referred as "transparency of locality"). Furthermore, the design of the distributed controllers

should separate clearly the design of the client and server role, in order to make it easier to redefine or completely change the communication protocol.

Solution

We use the Forwarder-Receiver design pattern [BMRS+95] to implement the transparent inter-process communication between the controllers. The client-side of the Controller is implemented by the forwarder component, and the server-side is implemented by the receiver component. This solution allows the controllers to be independent of the inter-process communication mechanism used, and guards them from details of the location of the other controllers. Figure 22 shows the design of the communication between the controllers using this pattern.

The forwarder is implemented by the abstract class ContClientInterface, which defines the methods of the client-side of the controller (i. e., the services that request services in other controllers). A concrete subclass of ContClientInterface defines an implementation for these methods, calling the services in the receiver object using a specific communication protocol. Depending on the protocol used, it performs previous routines needed to obtain a reference to the server and to call remote services (for instance, to marshal and to deliver the message to a remote server), and might also perform some treatment for communication faults.

The receiver is implemented by the abstract class ContServerInterface, which defines the methods that are provided remotely. A concrete subclass of ContServerInterface provides the implementation of the remote services, calling the respective methods implemented by the Controller. Depending on the protocol used, it performs some routines for receiving the remote request (for instance, to unmarshal messages received from remote forwarders, etc).

The topology of the separated boards determines the topology of the computer network, as can be observed in Figure 7 of section 3.1. A controller of one board can communicate with either one or two remote controllers that control the other boards. Then, the ContClientInterface has references for two possible remote receivers of the ContRemoteInterface class, and decides at runtime which one it should use. The Controller class is independent of these details.

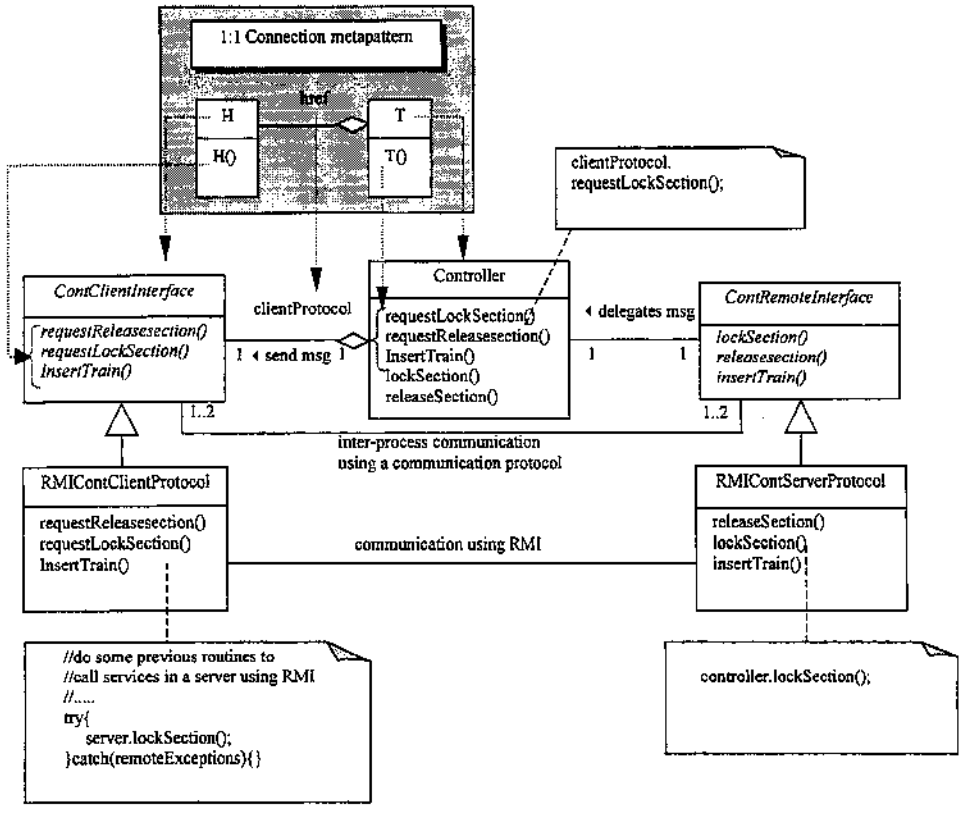


Figure 22: The design of the communication protocol using the Forwarder-Receiver pattern

The controller is configured with a concrete subclass of ContClientInterface and uses this specific forwarder to request services in other controllers. The concrete subclass of ContClientInterface and ContRemoteInterface implements the same specific protocol. The concrete subclass of ContRemoteInterface has a reference for the correspondent Controller object, and delegates to it the execution of the services that was requested remotely.

We have implemented the concrete classes RMIContClientProtocol and RMIContServerProtocol to implement the forwarder and receiver using the RMI (Remote Method Invocation) protocol provided by the Java programming language. Using this specific protocol, the receiver is also implemented as the Proxy design pattern. The RMIContClientProtocol has a reference for the skeleton of a RMIContServerProtocol that is the remote proxy of the actual RMIContServerProtocol object in another address space. The RMI protocol implements all the routines to marshal and unmarshal messages, so the communication between the forwarder and the receiver is very simple. The RMIContClientProtocol has to implement only some initializations of the proxy, and the RMIContServerProtocol has to be derived from a RemoteInterface interface (provided by the Java API) to provide the services remotely.

The 1:1 Connection metapattern is used to document this hot spot. The Controller is the template class that defines the template methods requestLockSection() requestReleaseSection() and insertTrain(). These methods call the hook methods defined by the hook class ContClientInterface, which should be redefined to implement a specific communication protocol.

4.3 Implementation issues

The dependable framework for train controllers has been implemented in the Java programming language. We have used a meta-object protocol called Guaraná [Oli98] to implement the MetaStateMachine at the meta-level. The complete framework implementation has approximately 110 classes and 7000 lines of codes.

We have also implemented an example-application using the specific configuration of the Train Set System in order to verify the level of adaptability achieved by the framework. The framework design was easily customized to accomplish the specific requirements, providing the desired level of reuse.

5 Conclusions

We have described the design of a framework for fault-tolerant train controller using architectural styles for the framework's architecture definition and the combination of design patterns and metapatterns for the hot spots descriptions. The main lessons learned about using architectural styles, patterns and metapatterns for the documentation of framework design are summarized below:

- The styles that we have applied in the framework architecture descriptions aim to provide fault tolerance mechanisms in a transparent way. The use of these architectural styles promotes a well-structured, less complex and more understandable architecture design.
- The use of architectural styles also provides a high-level design reuse, since they provide good solutions for the general system organization. Regarding the framework development, reusing these solutions has reduced the cost of the framework design.
- Most of the design patterns from pattern catalogs and pattern books document the detailed structure and semantic of the hot spots, helping framework's users to understand the hot spot design by describing the classes' responsibilities and the way the objects interact with each other to perform a required task. However, the design patterns do not communicate precisely

which part is fixed and which part should be redefined by a specific application. The metapatterns has been proposed as a means to document the flexible structures of other design patterns, identifying more precisely the fixed and adaptable parts of these design patterns. In this way, metapatterns and design patterns are complementary, and the use of both has improved the quality of the framework documentation.

- The design patterns have provided us with good solutions for complex hot spots problems, such as the hot spots for environmental fault tolerance and the hot spots for the communication protocol used to implement the distributed controllers. The reuse of these good solutions has reduced our development effort, allowing us to reuse the expertise of other developers in solving similar problems. Moreover, using well-know design patterns makes it easier to communicate the hot spot design to the framework's users and maintainers.
- Metapatterns are more abstract than design patterns, describing the variabilities without defining the detailed semantic of the hot spot design. They describe the relationships between fixed and variable parts, identifying the template and hook classes and their respective template and hook methods. Thus, the metapatterns document the parts that should (or have to) be changed when a framework user creates a specific application. However, the metapatterns cannot be considered as a "cookbook" that describes the steps to obtain a specific application, since they describe only what a framework's user is allowed to do (i. e. which class can be redefined, which method can be overridden) but they do not describe how it can be done (i.e. how to redefine a class and overridden its method to implement a specific feature).
- We have also realized some limitations of applying design patterns and metapatterns to describe the framework's hot spots. The number of design patterns has increased exponentially with the popularity of pattern's conferences, and the application developers cannot know the various existent patterns. Thus, the advantage of having "a common vocabulary to describe the solution for complex problems" cannot be guaranteed if the framework's users and maintainers do not know the patterns used in the design. Moreover, the use of an unknown pattern can even make it harder to understand the hot spot design, since we usually do not describe the pattern solution in details.
- Another limitation of design patterns and metapatterns is that they are described using existent object-oriented modeling languages, which do not provide appropriated notations for

the documentation of the variable and fixed parts. In this paper, we have used an "informal" notation for the representation of metapatterns (as suggested in the Pree's book [Pre95]). Using this informal representation of metapatterns, we cannot ensure that the framework's users will understand unambiguously the meaning of the metapatterns and what they are describing.

From the various advantages and some limitations of using design patterns and metapatterns we can draw the following conclusions: (i) design patterns and metapatterns are complementary, and they are a good solution for the hard problem of documenting a framework design; (ii) the lack of appropriated object-oriented design notations for describing flexible design makes the understanding of the metapatterns and design patterns more difficult, and some times ambiguous; (iii) the provision of appropriated notations and tools for representing the design patterns and metapatterns used to document the framework's hot spots is a step toward the construction of more understandable and reusable frameworks.

Bibliography

[BMA97] D. Brugali, G. Menga, A. Aarsten. The Framework Life Span. *Communications of the ACM* vol. 40, no. 10, pp. 65-68. October 1997.

[BMRS+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *A System of Patterns: Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996.

[Cl93] R. H. Campbell and N. A. Islam. A technique for documenting the framework of an object-oriented system. *Computing Systems* vol. 6, no. 4, 1993.

[Cli96] M. P. Cline. The Pros and Cons of Adopting and Applying Design Patterns in the Real World. *Communications of the ACM*, vol. 39, no.10, pp. 47-49. October 1996.

[Coa92] P. Coad. Object-Oriented Patterns. *Communications of the ACM.* vol. 33, no. 9. 1992.

[FR98a] L. L. Ferreira and C. M. F. Rubira, The Reflective State Pattern. *Proceedings of the 5th Pattern Languages of Programs Conference (PLoP'98)*, August 1998, Monticello, Illinois, USA.

[FR98b] L.L. Ferreira and C.M.F Rubira. Integration of Fault Tolerance

Techniques: a System of Pattern to Cope with Hardware, Software and Environmental Fault Tolerance. *Digest of FastAbstracts: FTCS'28 (the 28th Annual International Symposium on Fault-Tolerant Computing),* June 23-25, 1998, Munich, Germany, pp. 25-26

[FR98c] L. L. Ferreira and C. M. F.Rubira. Reflective Design Patterns to Implement Fault Tolerance. *Proceedings of the Workshop on Reflective Programming in C++ and Java: Workshop #13 of OOPSLA'98.* Vancouver, Canada, October, 1998. pp. 81-85.

[FS97] M. E. Fayad and D. C. Schmidt. Object-Oriented Application Frameworks. *Communications of the ACM* vol. 40, no. 10, pp. 39-42. October 1997.

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

[Joh97a] R. E. Johnson. Components, Frameworks, Patterns. *Proceedings of the 1997 Symposium on Software Reusability (SSR'97). Software Engineering Notes.* vol. 22, no. 3, pp. 10-15. May 1997.

[Joh97b] R. E. Johnson. Frameworks = Components + Patterns. *Communications of the ACM – Object-Oriented Application Frameworks.* vol. 40, no. 10, pp. 39-42. October 1997.

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

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

[Oli98] A. Oliva. *Guaraná: Uma Arquitetura de Software para Reflexão Computacional Implementada em Java.* Master Thesis. Institute of Computing, State University of Campinas, September 1998. http://www.dcc.unicamp.br/~oliva/guarana/

[Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley. 1995.

[Qua97] E. M. Quadros. *Uma abordagem Orientada a Objetos para Programação Distribuída Confiável.* Tese de Mestrado. Instituto de Computação – UNICAMP, maio de 1997.

[Roh95] H. Rohnert. The Proxy Design Pattern Revisited. *Pattern Languages of Program Design 2,* chapter 7, pp. 105 - 118. 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, October 1994.

[SB98] P. Sommerlad and F. Buschmann. Manager. *Pattern Languages of Program Design 3.* Editors: R. Martin, D. Riehle and F. Buschamann. Addison-Wesley, 1998.

[Sch96] H. A. Schmid. Creating Applications from Components: A Manufacturing Framework Design. *IEEE Software*, vol. 13, no. 6, pp. 67-75. November 1996.

[Sch97] H. A. Schmid. Systematic Framework Design by Generalization. *Communications of the ACM – Object-Oriented Application Frameworks.* vol. 40, no. 10, pp. 48-51. October 1997.

[Scht97] D. C. Schmidt. Applying design patterns and frameworks to develop object-oriented communication software. *Handbook of Programming Languages,* vol. 1, MacMillan Computer Publishing, 1997.

[SG96] M. Shaw and D. Garlan. Software Architecture, Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs, NJ, 1996.

[WG94] A. Weinand and E. Gamma. ET++ - a Portable, Homogenous Class Library and Application Framework. *In Proceedings of UBILAB Conference'94.* Universitatsverlag Konstanz, 1994.

Resumo do Capítulo 4

Neste capítulo, nós apresentamos o projeto detalhado do *framework*, utilizando estilos de arquitetura para o projeto de sua arquitetura, e padrões de projeto e metapadrões para a descrição dos seus pontos adaptáveis. As principais conclusões obtidas com o uso prático destas técnicas foram discutidas nas conclusões do artigo.

Nossa abordagem para o desenvolvimento do *framework* seguiu os seguintes passos: (1) a partir de um modelo de uma aplicação específica de controlador de trens, nós identificamos os pontos fixos e os pontos adaptáveis de acordo com a análise do domínio do problema; (2) aplicamos estilos de arquitetura para o projeto da arquitetura do *framework*, que define a funcionalidade comum do domínio, incluindo o requisito não-funcional de tolerância a falhas; (3) aplicamos uma seqüência de transformações no modelo de classes inicial utilizando padrões de projeto e metapadrões para a obtenção da adaptabilidade requerida; (4) implementamos o *framework* na linguagem de programação Java; (5) utilizamos o *framework* no desenvolvimento de uma aplicação específica do domínio, preenchendo os pontos adaptáveis de acordo com às características específicas da aplicação.

Um *framework* bem projetado e maduro implementa todas as funcionalidades do domínio e oferece a adaptabilidade requerida, de preferência na forma de componentes adaptáveis caixa-preta. Para a obtenção de um *framework* com estas características, são necessárias várias iterações da seqüência de passos acima. Nós executamos apenas a primeira iteração desta seqüência de passos e obtivemos um *framework* cuja adaptabilidade é obtida principalmente através de componentes caixa-branca, ou seja, através de derivação das classes de um ponto adaptável e redefinição de seus métodos. Seria necessária a implementação de outras aplicações específicas a partir do *framework* para verificar se os pontos adaptáveis satisfazem à variabilidade requerida por estas aplicações. No caso negativo, uma próxima iteração seria realizada, efetuando-se possíveis reestruturações no projeto e implementação do *framework*. Com um conhecimento mais profundo do domínio de controladores de trens, é possível obter também um projeto mais maduro do *framework*, implementando-se mais componentes caixa-preta, o que facilitaria sua reutilização.

Trabalhos relacionados:

Existem vários trabalhos relacionados ao desenvolvimento de *frameworks* que propõem metodologias, linguagens, técnicas e ferramentas que podem auxiliar no processo seu desenvolvimento. O principal objetivo destes estudos é a redução do custo de desenvolvimento, e ao mesmo tempo, o aumento da qualidade do projeto e consequentemente, do grau de reutilização obtido com o *framework.* Para a descrição dos pontos adaptáveis de *frameworks*, existem vários trabalhos que utilizam padrões de projeto no projeto de *frameworks,* entre eles, um *framework* para sistemas de manufatura [Sch96] e um *framework* para sistemas de comunicação [Sdt95]. Outros trabalhos utilizam a abordagem de metapadrões para descrever a adaptabilidade de outros padrões, como por exemplo, a documentação do framework ET++ em [Pre95] e um recente trabalho de mestrado que documenta um framework reflexivo para interface gráficas também utilizando padrões e metapadrões [Coe98]. Outra linha de pesquisa é a utilização de recursos de linguagens de modelagem para a documentação de *frameworks.* Catalysis [SW97] utiliza o recurso de *stereotype* da linguagem UML, e propõe um método de projeto para a construção de *frameworks.* M. Fontoura [Fon99] estende a notação UML para a descrição dos pontos adaptáveis do *framework* e propõe o uso de ferramentas que auxiliam no processo de seu desenvolvimento.

Capítulo 5

Conclusões e Trabalhos Futuros

Esta dissertação concentrou-se na utilização de técnicas de reutilização de software tais como estilos de arquitetura, padrões de projeto e metapadrões para o desenvolvimento de um *framework* orientado a objetos para o domínio de controladores de trens tolerantes a falhas e distribuídos. Durante o desenvolvimento do *framework*, chegamos a vários resultados diretos e indiretos, que formam as principais contribuições deste trabalho. As principais contribuições são:

- Projeto de um *framework* orientado a objetos para o domínio de controladores de trens tolerantes a falhas e distribuídos. Utilizamos estilos de arquitetura para a documentação da arquitetura do *framework*, e padrões de projeto e metapadrões para a documentação dos seus pontos adaptáveis. Com a utilização destas técnicas, obtivemos um projeto mais estruturado e fácil de ser entendido, mantendo-se a complexidade sob controle.
- Implementação do *framework* na linguagem Java, utilizando a arquitetura reflexiva do Guaraná para a implementação dos componentes reflexivos. Nós também implementamos uma aplicação com configurações específicas reutilizando o *framework*, com o objetivo de analisar o grau de reutilização obtido. Como conclusão, podemos dizer que a arquitetura do *framework* implementa a funcionalidade comum do domínio da aplicação, e oferece os pontos adaptáveis adequados para a implementação das características de aplicações específicas.
- Documentação do padrão "*Reflective State*" e um sistema de padrões formado pelas variações deste padrão para o domínio de tolerância a falhas. Estes padrões utilizam reflexão computacional para implementar os aspectos de controle relacionados à implementação de requisitos não-funcionais, de forma separada e transparente para os objetos da aplicação. A utilização de uma variação do padrão *Reflective State* no projeto dos componentes tolerantes a falhas do *framework* ofereceu uma solução mais flexível e fácil de ser estendida do que a solução do padrão original.

- Implementação do padrão *Reflective State* utilizando a arquitetura do Guaraná. Esta implementação representa também um *framework* para implementação de máquinas de estados. Este *framework* implementa as classes abstratas e concretas que representam a máquina de estados no meta-nível, as quais podem ser configuradas e/ou estendidas para a implementação de uma máquina de estados específica.
- Documentação do novo estilo de arquitetura denominado *Idealized Fault-Tolerant Component* que oferece uma solução para a estruturação de sistemas tolerantes a falhas baseado no modelo de mesmo nome proposto inicialmente por Lee e Anderson [LA90]. A documentação deste modelo como um estilo de arquitetura permite a reutilização da solução por outros arquitetos de software na definição de arquiteturas de sistemas tolerantes a falhas.

Trabalhos Futuros

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

- Implementação de uma ferramenta para a geração do código de uma máquina de estados específica que reutiliza a máquina de estados genérica definida pelo padrão *Reflective State* (*framework* para máquina de estados). Este código seria gerado automaticamente a partir de um diagrama de estados definido para uma classe da aplicação. Como os aspectos de controle da máquina de estados são implementados separadamente das classes da aplicação, a ferramenta poderia implementar também as futuras extensões e modificações realizadas na máquina de estados, sem alterar o código das classes da aplicação. Isto facilitaria as extensões do *framework* no que diz respeito aos componentes tolerantes a falhas.
- Implementação do mecanismo de tratamento de exceções dos componentes tolerantes a falhas do *framework* utilizando uma arquitetura reflexiva, como foi proposta em [GBR99]. Como foi discutido no capítulo 3, esta solução define os tratadores de exceções (parte anormal do componente) em uma classe separada, e utiliza um mecanismo de tratamento de exceções reflexivo para a localização e execução destes tratadores.
- A análise das medidas dos *overheads* causados pelo uso de reflexão computacional na implementação dos componentes tolerantes a falhas que utilizam o padrão *Reflective State*. Para uma análise comparativa, seria necessário também a implementação de uma versão não reflexiva dos componentes tolerantes a falhas utilizando o padrão *State* original.

- Desenvolvimento de um *cookbook* para guiar a reutilização do *framework* no desenvolvimento de aplicações específicas. Uma alternativa seria também o desenvolvimento de um diagrama de instanciação, que é um cookbook formal, podendo-se assim construir uma ferramenta para a instanciação do framework.
- Amadurecimento do *framework* através da execução de mais algumas iterações da seqüência de passos apresentadas no capítulo 4. Seria necessária a implementação de outras aplicações específicas a partir do *framework*, para verificar se os pontos adaptáveis satisfazem à variabilidade requerida por estas aplicações. No caso negativo, uma próxima iteração seria necessária, realizando-se possíveis reestruturações no projeto e implementação do *framework*. Com um conhecimento mais profundo do domínio de controladores de trens, é possível obter também um projeto mais maduro do *framework*, implementando-se mais componentes caixa-preta, o que facilitaria sua reutilização.

Referências Bibliográficas

[BMRS96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *A System of Patterns: Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996.

[BRL97] L.E. Buzato, C.M.F. Rubira, e 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.

[Coe98] M.G. Coelho. *Uma abordagem Reflexiva para a Construção de Frameworks para Interfaces Homem Computador.* Tese de mestrado do Instituto de Computação da Unicamp, Campinas, novembro de 1998.

[Fon99] M.F.M.C. Fontoura. *A Systematic Approach to Framework Development.* Tese de doutorado. Departamento de Ciência da Computação – Universidade Católica do Rio de Janeiro, julho de 1999.

[FPB95] J.-C. Fabre, T. Pérennou, e L. Blain. Meta-object Protocols for Implementing Reliable and Secure Distributed Applications. *Technical Report LASS-95037*, Centre National de la Recherche Scientifique, fevereiro de 1995.

[FR98a] L.L. Ferreira e C.M.F. Rubira, The Reflective State Pattern. *Proceedings of the 5th Pattern Languages of Programs Conference (PLoP'98)*, agosto de1998, Monticello, Illinois, USA. Technical report # WUCS-98-25.

[FR98b] L. L. Ferreira e C. M. F.Rubira. Reflective Design Patterns to Implement Fault Tolerance. *Proceedings of the Workshop on Reflective Programming in C++ and Java: Workshop #13 of OOPSLA'98.* Vancouver, Canada, Outubro de 1998. pp. 81-85.

[FR98c] L.L. Ferreira e C.M.F. Rubira. Padrão State Reflexivo: Refinamento do Padrão de Projeto State para uma Arquitetura Reflexiva. *Anais do XII Simpósio Brasileiro de Engenharia de Software (SBES'98)*, Maringá, Paraná, outubro de 1998.

[FR98d] L.L. Ferreira e C.M.F. Rubira. Integration of Fault Tolerance Techniques: a System of Pattern to Cope with Hardware, Software and Environmental Fault Tolerance. *Digest of FastAbstracts: FTCS'28 (the 28th Annual International Symposium on Fault-Tolerant Computing)*, Monique, Alemanha, pp. 25-26, junho

de1998

[GBR99] 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. *Submetido para IEEE 10th International Symposium on Software Reliability Engineering*, Boca Raton, Florida, novembro de 1999.

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

[JF88] R.E. Johnson e B. Foote. Designing Reusable Classes. *Journal of Object Oriented Programming – JOOP.* vol 1, no. 22, pp. 22-35, junho de 1988.

[Joh97a] R. E. Johnson. Components, Frameworks, Patterns. *Proceedings of the 1997 Symposium on Software Reusability (SSR'97). Software Engineering Notes.* vol. 22, no. 3, pp. 10-15, maio de 1997.

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

[Lis98] M.L.Lisboa. A New Trend on the Development of Fault-Tolerant Applications: Software Meta-Level Architectures. *Proceedings of the 1998 IFIP – International Workshop on Dependable Computing and its Applications.* Johannesburg, África do Sul, janeiro de1998.

[Oli98] A. Oliva. *Guaraná: Uma Arquitetura de Software para Reflexão Computacional Implementada em Java.* Tese de Mestrado. Instituto de Computação, Unicamp. Campinas, SP, setembro de 1998. http://www.dcc.unicamp.br/~oliva/guarana/

[Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley. 1995.

[RBP+92] J. Rumbaugh, M. Blaha, W. Premerlani,F. Eddy e W. Lorense. *The Object-Oriented Modeling and Design.* Prentice-Hall International, Inc., Englewood, Cliffs, NJ, EUA. 1992.

[RX93] B. Randell e J.Xu. Object-Oriented Software Fault Tolerance: Framework, Reuse and Design Diversity. *PDCS2 First Year Report,* Predictably Dependable Computing Systems, 1: 165-184, Toulose, França, setembro de 1993.

[Sch96] H. A. Schmid.Design Patterns for Constructing the Hot Spots of a Manufactoring Framework. *Journal of Object-Oriented Programming – JOOP*, junho de 1996.

[Sdt95] D.C. Schmidt. Experience Usign Design Patterns to Develop Reusable Object-Oriented Communication Software. Communication of the ACM, vol. 38, no. 10, outubro 1995.

[SG96] M. Shaw e D. Garlan. Software Architecture, Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs, NJ, 1996.

[Str92] R. Stroud. Transparency and Reflection in Distributed Systems. In *5th European SIGOPS Workshop on Models and Paradigms for Distributed Systems Structuring*, Mont Saint-Michel, França, setembro de 1992. ACM SIGOPS, IRISA, INRIA-Rennes.

[SW94] R. J. Stroud e Z. Wu. Using Meta-Objects to Adapt a Persistent Object System to Meet Applications needs. *In 6th SIGOPS European Workshop on Matching Operating Systems to Applications Needs*. 1994.

[SW97] D. D'Souza and A.Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison Wesley, 1997.

[YTT89] Y. Yokote, F. Teraoka e M. Tokoro. A Reflective Architecture for an Object-Oriented Distributed System. In *Proceedings of ECOOP'89*, 1989.