UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Domain-Oriented Reuse Interfaces for Object-Oriented Frameworks

André Leal Santos

DOUTORAMENTO EM INFORMÁTICA
ESPECIALIDADE ENGENHARIA INFORMÁTICA

2008

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Domain-Oriented Reuse Interfaces for Object-Oriented Frameworks

André Leal Santos

DOUTORAMENTO EM INFORMÁTICA
ESPECIALIDADE ENGENHARIA INFORMÁTICA

2008

Tese orientada pelo Prof. Doutor Kai Evert Ross Koskimies
e co-orientada pela Prof.^{a} Doutora Maria Antónia Bacelar da Costa Lopes

Abstract

Object-oriented frameworks play an important role in current software engineering practice. Frameworks enable the development of several applications in a particular domain with high levels of reuse. However, although frameworks are a powerful means for software reuse, their *reuse interface* is typically difficult to learn, a problem which hinders their full potential. Different strategies can be used to overcome this problem, namely by developing documentation, instantiation tools, or a *domain-specific language* (DSL). Although DSLs provide the most effective support for using a framework, developing and maintaining them are challenging and laborious tasks.

This work proposes a technique based on *aspect-oriented programming* for enhancing framework reuse interfaces, so that framework-based applications can be developed at a higher abstraction level. A pattern language for aiding the design of such reuse interfaces is also presented. Although the adoption of higher level reuse interfaces facilitates framework usage, this work goes one step further and proposes a technique that, capitalizing on such reuse interfaces, enables automation gains on the development of DSLs for instantiating frameworks. By exploiting the close relation between application concepts and code based on the proposed reuse interfaces, ready-to-use DSLs can be automatically extracted using a generic *language workbench*. A prototype of such language workbench for building *domain-specific modeling languages* has been implemented, and the proposed techniques have been evaluated using two real frameworks.

Keywords: Object-Oriented Frameworks, Aspect-Oriented Programming, Domain-Specific (Modeling) Languages, Language Workbenches.

Resumo

As *frameworks* orientadas a objectos assumem um papel importante na prática de Engenharia de Software, pois permitem desenvolver diferentes aplicações num determinado domínio, com elevados níveis de reutilização. Contudo, embora as *frameworks* sejam um meio poderoso para reutilização de software, tipicamente a sua *interface de reutilização* é difícil de dominar — um problema que limita o seu potencial. Diferentes estratégias podem ser adoptadas de forma a ultrapassar este problema, nomeadamente a produção de documentação, a utilização de ferramentas de instanciação, e a definição de DSLs. Embora a última opção seja especialmente poderosa, concretizar e manter uma DSL tem tipicamente um custo elevado.

Este trabalho de Doutoramento propõe uma técnica baseada em *programação orientada a aspectos* para definir as interfaces de reutilização das *frameworks* de forma a facilitar a sua instanciação, permitindo que as aplicações sejam desenvolvidas a um nível de abstracção mais elevado. É também apresentada uma linguagem de padrões para auxiliar o desenvolvimento de tais interfaces de reutilização. Este trabalho explora a automação do desenvolvimento de DSLs para instanciação de frameworks, tirando partido da proximidade entre os conceitos de uma aplicação baseada na *framework* e as interfaces de reutilização propostas. É proposta uma abordagem na qual DSLs prontas a usar são extraídas das interfaces de reutilização, por via de uma ferramenta genérica. As abordagens foram avaliadas utilizando duas *frameworks* reais, e foi concretizado um protótipo da ferramenta proposta.

Palavras Chave: *Frameworks* Orientadas a Objectos, Programacao Orientada a Aspectos, Linguagens Especificas de Dominio

Resumo Alargado

Uma *framework* orientada a objectos consiste num conjunto de classes que implementa uma solução abstracta para aplicações de determinado domínio. Desde há vários anos que as *frameworks* assumem um papel importante na prática de Engenharia de Software, particularmente na concretização de interfaces gráficas, *middleware*, e linhas de produto de software. Utilizando uma *framework*, uma aplicação é desenvolvida adaptando determinados aspectos da solução abstracta a requisitos específicos. A adaptação consiste numa instanciação da *framework* definida em termos dos elementos disponíveis na interface de reutilização. As interfaces de reutilização são tidas como difíceis de aprender, e é reconhecido que os programadores de aplicações podem levar meses a saber instanciar correctamente uma *framework* não trivial. Uma *framework* é tipicamente composta por um número elevado de classes, o que dificulta a identificação das partes que é necessário adaptar. Os padrões de colaboração entre os objectos de uma aplicação baseada na *framework* são também tipicamente difíceis de aprender, dado que requerem uma compreensão parcial do *design* da *framework*.

A dificuldade em dominar a interface de reutilização de uma *framework* torna necessário disponibilizar artefactos externos à *framework* que dão suporte ao desenvolvimento de aplicações. O tipo de suporte mais popular é baseado em documentação, tipicamente estruturada em forma de "livro de receitas". Cada receita explica como adaptar determinado aspecto de uma aplicação, normalmente com o auxílio de um exemplo. Apesar da documentação auxiliar o programador de aplicações, não o dispensa de lidar com a complexidade da interface de reutilização.

Uma abordagem mais poderosa do que documentação "em papel" consiste em desenvolver ferramentas de instanciação. Estas podem assumir diversas formas, que vão desde documentação interligada que guia o programador no processo de instanciação, a *wizards* que permitem a geração de partes do código das aplicações. Este tipo de ferramentas pode esconder parte da complexidade da interface de

reutilização da *framework*, mas tipicamente não permite gerar todo o código fonte de uma aplicação.

O suporte mais poderoso para utilização de uma *framework* é alcançado por meio de uma linguagem específica de domínio (DSL, *Domain Specific Language*). Utilizando uma DSL, as aplicações são descritas a alto nível com base em abstracções do domínio da *framework*. A descrição de uma aplicação utilizando a DSL permite gerar o código fonte que instancia a *framework* através da sua interface de reutilização. Desta forma, a complexidade da interface de reutilização é escondida do programador de aplicações, o qual nalguns casos pode apenas ser um especialista de domínio sem conhecimentos de programação. A adopção de DSLs é tida como uma estratégia que permite obter elevados ganhos na produtividade e qualidade no desenvolvimento de aplicações.

Embora o poder de uma DSL para utilização de uma *framework* seja atractivo, diversas dificuldades podem estar associadas à sua concretização. Uma das principais fontes de tais dificuldades é a complexidade da transformação das abstracções descritas numa DSL em código fonte baseado na interface de reutilização. A forma mais comum de concretizar tal transformação é por via de um gerador de código. O principal obstáculo na obtenção de uma solução de DSL bem sucedida está portanto em grande parte relacionado com a concretização e manutenção do gerador de código. A manutenção assume uma relevância essencial, dado que as *frameworks* tipicamente evolvem constantemente devido à evolução do domínio das mesmas. O suporte para utilização de novas funcionalidades da *framework* através da DSL, implica modificar a definição das abstracções da DSL e o gerador de código.

O trabalho de doutoramento apresentado nesta tese aborda a dificuldade de utilização de *frameworks* sobre duas perspectivas. Uma primeira abordagem permite desenvolver interfaces de reutilização de *frameworks* que permitem a sua instanciação a mais alto nível. Tal abordagem facilita a utilização de *frameworks* dada a elevação do nível de abstracção. Por outro lado, uma segunda abordagem baseada na primeira, permite definir uma DSL na própria interface de reutilização. Desta forma, a DSL é concretizada somente na interface de reutilização, dispensando a necessidade de desenvolver um gerador de código.

A primeira abordagem é baseada no conceito de aspectos de especialização de *frameworks.* Utilizando programação orientada a aspectos, é desenvolvido um aspecto de especialização para cada conceito da *framework.* Um aspecto de especialização é um módulo reutilizável que dá suporte à instanciação de um conceito numa aplicação. Um aspecto de aplicação é um módulo que utiliza um aspecto de especialização para instanciar um conceito numa aplicação. Um aplicação é composta por diversos aspectos de aplicação, cada um dos quais encapsulando a utilização de um conceito oferecido pela *framework.* Em comparação com a instanciação convencional, esta abordagem permite construir soluções modulares, nas quais os programadores de aplicações não necessitam de dominar tantos detalhes da *framework.* É demonstrado que a linguagem AspectJ é adequada para desenvolver aspectos de especialização para *frameworks* concretizadas em Java.

Dado que o desenvolvimento de aspectos de especialização não é trivial, foi elaborada uma linguagem de padrões para auxiliar nesta tarefa. Tal linguagem é composta por vários padrões de desenho interligados, descrevendo soluções recorrentes no desenho de aspectos de especialização.

A segunda abordagem é baseada no conceito de interface de reutilização orientada ao domínio (DORI, *Domain-Oriented Reuse Interface*). Tal interface de reutilização é composta por aspectos de especialização, os quais exprimem abstracções de uma DSL. Uma DSL é extraída da DORI utilizando um ferramenta adicional, e está pronta a ser utilizada para desenvolver aplicações baseadas na *framework.* A ferramenta é genérica, no sentido em que pode ser utilizada com qualquer *framework* para a qual exista uma DORI desenvolvida na linguagem de programação orientada a aspectos suportada. Ao desenvolver uma DORI para determinada *framework*, uma DSL é obtida automaticamente, não sendo necessário desenvolver quaisquer artefactos adicionais.

A técnica proposta foi concretizada num protótipo denominado ALFAMA (*Automatic DSLs for using Frameworks by combining Aspect-oriented and Metamodeling Approaches*), o qual suporta a construção de DSLs de modelação onde as aplicações são descritas através de um grafo. A ferramenta ALFAMA é baseada na plataforma Eclipse, e as DSLs são definidas utilizando a tecnologia de modelação EMF (*Eclipse Modeling Framework*).

As abordagens propostas foram elaboradas e validadas utilizando duas *frameworks* como casos de estudo. As *frameworks* foram o *JHotDraw*, que permite desenvolver aplicações para edição de gráficos estruturados, e o *Eclipse RCP (Rich Client Platform)*, que permite desenvolver aplicações independentes baseadas na infra-estrutura gráfica e arquitectura do Eclipse.

A forma como as *frameworks* são concretizadas pode-se considerar consolidada de alguns anos a esta data. No entanto, investigação focada no suporte de utilização é actualmente activa, dado que a eficácia dos métodos e técnicas existentes não é tida como satisfatória. O trabalho apresentado nesta tese lança uma nova perspectiva sobre as interfaces de reutilização de *frameworks*. O novo tipo de interfaces de reutilização proposto não só contribui para facilitar a utilização da *framework* por via de programação, como permite a concretização de DSLs com um menor custo. A existência de DSLs é benéfica para a prática de Engenharia de Software, dado o potencial aumento de produtividade e qualidade que pode ser alcançado. As abordagens propostas nesta tese contribuem no sentido de melhorar as técnicas e infra-estrutura de suporte ao desenvolvimento de DSLs.

Acknowledgements / Agradecimentos

Kiitos paljon Kai, for the good and dedicated advising that you gave me throughout the Ph.D. studies. From the moment I started to apply for scholarship, until the thesis delivery, I feel that you always did the best you could for helping me. Moreover, you were the one who made possible for me to be a visiting researcher at Tampere University of Technology during two years. Without such a possibility, I guess I would have not followed the Ph.D. studies.

Thanks to the members of the research group Practise (Tampere University of Technology) for their collaboration, with a special *shukran* to Imed Hammouda for his encouragement and support.

Muito obrigado Antónia, não só pela orientação do Doutoramento, mas também pelo incentivo desde cedo para desenvolver as minhas capacidades de investigação, tendo todavia, sempre em conta que há vida para além dos *papers*. Sempre que andei meio perdido nos dilemas da vida académica, a ajuda foi preciosa.

Obrigado aos colegas, amigos, e professores do DI/FCUL que me apoiaram ou ajudaram de alguma forma durante o Doutoramento. Um agradecimento especial para Daniel Gomes, Bruno Martins, Mário Silva, Vasco Vasconcelos, Marcirio Chaves, João Antunes, Mônica Dixit, e Giuliana Santos. Muito obrigado ao Ademar Aguiar da FEUP, pelo apoio, confiança, e ajuda nas actividades do meu Doutoramento.

Obrigado à Fundação para a Ciência e Tecnologia, e consequentemente ao Estado Português, pelo financiamento desta actividade de investigação através de Bolsa Doutoramento (com identificação SFRH/ BD/ 16801/ 2004). Vou fazer o meu melhor para que o investimento tenha o seu retorno.

Thanks to Prof. Oscar Nierstrasz, the main opponent of this thesis' defense, for his valuable suggestions of improvement, which were taken into account in the post-defense version of this thesis.

Obrigado meus amigos e amigas por não me fazerem esquecer que **a vida não é só bytes**. Obrigado Tito e Mata, por entre muitas outras coisas, acreditarem em mim desde dos primeiros anos do curso. *Gracies* Josep, **obrigado** Albino, e *xièxie* Pan, pelo apoio, galhofas, e tudo mais. Obrigado Tatiana por todo o carinho e preocupação, e por conseguires ter feito com que me sentisse bem comigo mesmo.

Obrigado Miguel *Musta* **pelos bons momentos** da vida finlandesa com um travozinho português que levámos. Obrigado Ana pelo teu amor e paciência durante o tempo que estive fora. *Kiitos* Jaakko for helping me to get along well in Finland. *Spaciba balshoe* Petr for your major contribution for my happy life during the years in Tampere, I would have returned home earlier if it wasn't for you.

Estou-vos eternamente grato minha família. Obrigado mano Miguel, avô Domingos, e tio Jorge, pela vossa ajuda e consideração. Obrigado vovós Ivone e Claudina **por** sempre me tratarem como se eu fosse o vosso menino pequenino. Muito obrigado mãe Lena, tu que representas melhor que ninguém **aquele apoio que nunca faltou**.

à mãe Lena,

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the context of this dissertation in Section 1.1. Section 1.2 describes the problems addressed by the work of this dissertation. Section 1.3 summarizes the proposed solution to the problems, while Section 1.4 enumerates the main contributions. Finally, Section 1.5 outlines the remaining chapters of this dissertation.

1.1 Context

A state-of-the-practice approach for achieving a reusable software design together with its implementation is to build an *object-oriented framework* (Johnson & Foote, 1988), hereinafter, simply *framework*. A framework is typically associated with a certain domain, which can have a narrow scope (e.g. *software product-line* (SEI, 2008)), or a wider one (e.g. GUI framework). Frameworks are a powerful and established means for building reusable software (Fayad *et al.*, 1999) that allow a set of related applications to have a common implementation of their architecture and shared functionality. For instance, software product-lines apply framework-based development for achieving large-scale reuse (Bosch, 2000).

A *framework-based application* is an application that is built using a given framework. The set of framework elements that have to be known by application developers in order to build applications is referred to as the framework's *reuse interface*, while the process of building a framework-based application is referred to as *framework instantiation*.

A powerful way to support framework usage is to develop a *domain-specific language* (DSL), which enables framework-based applications to be generated from high-level descriptions that are expressed using domain concepts. By adopting this approach, application code is generated from *application models*, instead of being manually coded using the framework classes directly. The term "F-DSL" is used throughout this dissertation for referring to a DSL that generates applications based on a given framework. F-DSL solutions have a major impact in terms productivity. Application developers are relieved of understanding both the framework and its implementation language, and they are able to concentrate only on domain complexity. In fact, the users of an F-DSL may even be domain experts that are not programmers.

An F-DSL is used to describe application models, which are the input for generating framework-based applications. Therefore, an F-DSL has to embrace all the concepts that may be involved in all possible applications based on the framework. The definition of the concepts of an F-DSL can be given in terms of a *domain variability model* (Pohl *et al.*, 2005). A domain variability model describes the *variability* offered by the framework, i.e. where variation occurs within the set of applications (*variation points*), and which are the possible variable elements (*variants*) (Gurp *et al.*, 2001). The domain variability model can be thought of as being a *meta-model* that defines the modeling language in which application models are described.

In order that an F-DSL can be a *generative language*, it must have an associated *transformation definition*, which accepts application models as input and outputs code that instantiates the framework. Using the terminology associated with generative software development (Czarnecki & Eisenecker, 2000), Figure 1.1 depicts framework-based development using F-DSLs. On the one hand, domain abstractions are part of the *problem space*, i.e. the domain variability model and the application models. On the other hand, implementation abstractions are part of the *solution space*, i.e. the framework and the framework-based applications. A domain variability model sets the abstract syntax of the F-DSL, while the transformation defines its semantics. The concrete syntax depends on the language technology that is used (e.g. EMF (Eclipse Foundation, 2007d), UML Profiles (OMG, 2004), XML (W3C, 2008a), attribute grammars). The transformation

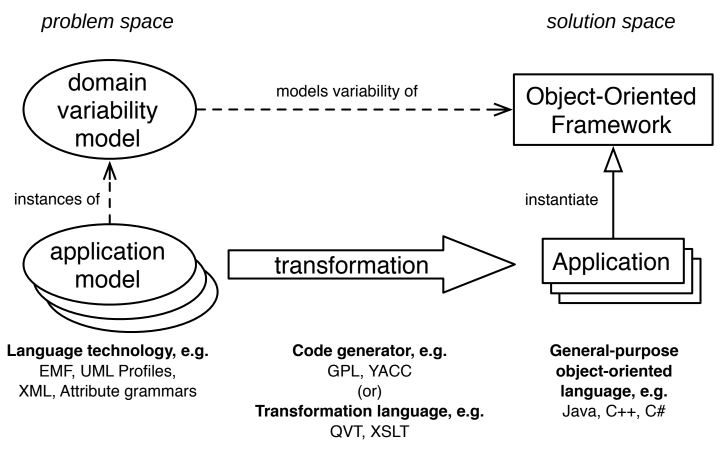


Figure 1.1: Generation of framework-based applications using DSLs.

is typically realized by a code generator written in a general-purpose language (GPL) or using a compiler generator (such as YACC (Johnson, 1979)). Another less common option is to realize the transformation using a transformation language associated with the adopted language technology, such as XSLT (W3C, 2008b) in the case of XML, or QVT (OMG, 2006) for EMF-like technologies. Transformation languages are intended to be a high-level and more elegant mechanism to define the transformations, but the immaturity or lack of expressiveness often makes these languages difficult to use in for model-to-code transformations.

1.2 Problems

The adoption of F-DSL solutions is in the first place motivated by the difficulty of using frameworks. The reason for this difficulty is mainly due to the complexity of framework reuse interfaces. However, although F-DSLs are capable to hide the complexity of reuse interfaces, their realization also has its own difficulties. Such difficulties constitute an obstacle in the development of F-DSLs.

1.2.1 Complexity of framework reuse interfaces

Learning a framework's reuse interface is recognized as a difficult and time-consuming task (Moser & Nierstrasz, 1996). In order that an application developer can successfully use a framework, he or she has to master its reuse interface. Fully mastering the reuse interface of a large and complex framework can take months. The main obstacles are related with the identification of the relevant framework elements for implementing a certain aspect in a framework-based application and understanding the collaborations between the objects of an application. The identification of the relevant framework elements is difficult both due to the large number of framework classes and the fact that typically only a fraction of the elements of a framework class is relevant for building an application. Understanding the collaborations between objects is difficult because one has to understand parts of the framework design, at least to a certain extent.

The difficulties related with learning hinder the successful adoption of frameworks. Due to this problem, application developers are usually supplied with other development artifacts in addition to the framework classes, such as documentation (e.g. Johnson, 1992; Krasner & Pope, 1988), instantiation tools (e.g. Fairbanks *et al.*, 2006; Hakala *et al.*, 2001; Ortigosa *et al.*, 2000), or an F-DSL (e.g. Durham & Johnson, 1996; Roberts & Johnson, 1997). Among these different strategies for overcoming the problem of framework learning, the F-DSL approach is the most effective given that it raises the abstraction level to domain concepts, hiding the complexity of a framework's reuse interface. However, although F-DSLs are appealing from an application development viewpoint, they may be difficult to realize.

Despite the existing solid technology for building DSLs (e.g. Eclipse Foundation, 2007d; Johnson, 1979; MetaCase, 2008), F-DSLs are not widespread. Given the attractiveness of F-DSLs, this can be considered as a hint that suitable solutions — while cost-effective — are difficult to build and maintain. There are two fundamental issues related to the development of an F-DSL: the abstraction level of the language concepts and the complexity of the mapping between concepts and framework elements. These issues have a big impact on the required effort to implement and maintain the code generator (or equivalent). In order to successfully build an F-DSL, the trade-offs associated with these two issues have to be well-balanced.

1.2.2 Defining DSL concepts

When building an F-DSL, the first issue that arises is to know exactly which concepts should be included in the language. Ideally, the concepts of an F-DSL should resemble as much as possible “pure” domain concepts (i.e. they should not be related with implementation issues), in order to have an effective rise in the abstraction level. However, this is not always the case. At two extremes, the definition of the concepts of an F-DSL may follow either a *domain-oriented* or an *implementation-oriented* approach.

In case a domain-oriented approach is followed, the concepts are defined according to the actual domain concepts without taking into account the framework implementation. In this case, the code generator logic is likely to be complicated, due to mismatches between the concepts and framework elements. Although this is the most convenient approach from an application development viewpoint, the complexity of the code generator may compromise a successful implementation and evolution of the F-DSL.

In case an implementation-oriented approach is followed, the concepts are closely tied to the framework elements, implying that the F-DSL will have concepts that are not purely domain-related. In this case, the code generator logic tends to be more simple, given that the transformation is more straightforward (Pohjonen & Tolvanen, 2002). However, although application developers might be more productive when using an implementation-oriented F-DSL instead of using the framework directly, it is more beneficial to have a language with a higher abstraction level (Kelly & Tolvanen, 2008). In order that the full potential of F-DSL solutions can be reached, the F-DSLs should be as domain-oriented as possible.

Having the F-DSL concepts defined, these have to be mapped to framework elements, so that framework-based code can be generated from an application model.

1.2.3 Mapping DSL concepts to framework elements

In the object-oriented programming paradigm, concepts are intended to have a direct correspondence to well-identified implementation elements — the classes.

In turn, concept instances are represented in objects which are parameterized and composed. However, in general, the reuse interface of a framework is not structured according to the concepts that can be instantiated in a framework-based application. In order to instantiate a concept in an application, mechanisms other than class instantiation may have to be used, namely, inheritance and hook method overriding. If the reuse interface would only be composed by classes representing the domain concepts of an application, the transformation of the concept instances of an application model into framework-based code would be straightforward.

Following a domain-oriented approach enables the development of F-DSLs with a higher abstraction level than if following an implementation-oriented approach. However, the higher the abstraction level of the language concepts is, the bigger the gap between concepts and framework elements, and in turn, the greater the complexity of the transformation.

An issue that significantly affects the complexity of the transformation is the many-to-many mapping between the concept instances of an application model and the framework-based code modules that represent those concept instances. The transformation of a single concept instance may affect more than one class, while the transformation of several concept instances may affect a single code module. In the general context of translating requirements to design and implementation, the first phenomenon is known as *scattering*, whereas the latter is known as *tangling* (Kiczales *et al.*, 1997). The case of transforming application models is similar, but the translation is automated by the transformation instead of being manually performed. This issue is depicted in Figure 1.2. We can see an example of scattering in the transformation of concept instance C3, which results in code module M2 and a part of code module M1. On the other hand, we can see an example of tangling in the transformation of concept instances C1, C2, and C3, which all affect code module M1.

In principle, raising the abstraction level of the reuse interface of the framework would shorten the gap between concepts and code, which would contribute to a less complex transformation. However, object-oriented programming languages are limited with respect to this goal because they do not enable to modularize the instantiation of certain concepts. On the other hand, the transformation

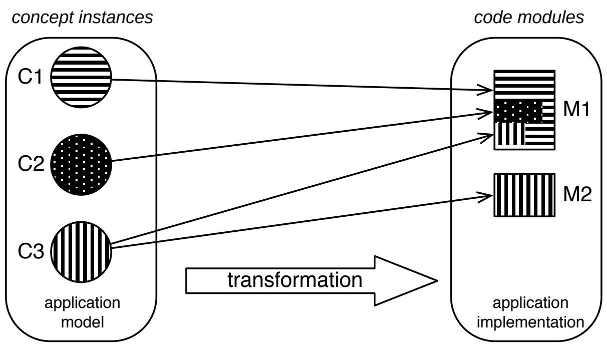


Figure 1.2: Scattering and tangling in model-to-code transformations.

could be simplified by lowering the abstraction level of concepts, so that they relate more directly to framework elements. However, as explained, this option is not optimal with respect to F-DSL effectiveness.

1.2.4 Building and maintaining code generators

The transformation of concept instances into framework-based code is typically realized by a code generator. The difficulty of building and maintaining the code generator is related to the trade-offs associated with the abstraction level of F-DSL concepts and their mapping to framework-based code. The many-to-many nature of this mapping affects the transformation complexity, given that it implies handling issues related to the generation of interleaved code that pertains to different concepts. The main difficulty resides in developing and maintaining a program that generates scattered and tangled code.

Besides the issue of the transformation complexity, code generators have other characteristics that make the task of implementing them more difficult. One is their “meta-nature”. A code generator is a program that generates another program. This characteristic seriously hinders comprehension, and therefore, the evolution of code generators. Another characteristic is their fragility. The generator outputs statements of a certain programming language. The identifiers of the elements of a reuse interface are manipulated as strings and treated in a non-compilable way — e.g. a misspelled method name on these statements does not indicate a compile error. Therefore, code generators can “silently” become broken when the reuse interface is modified.

Finally, the need of integrating manual code with generated code may be challenging. This might be necessary due to domain instability or the need of having *open variation points* (Gurp *et al.*, 2001) — i.e. certain variations in applications that cannot be predefined. Mechanisms for enabling this are referred to as *escapes* (Cleaveland, 1988). Ideally, it is not intended that the application developer has to manipulate or even understand code that is generated, in analogy with the case of general-purpose programming languages — the assembly language or bytecode are not meant to be understood by programmers. Therefore, there should exist mechanisms to integrate manual code that do not require inspection or manipulation of generated code. One of the main difficulties is to allow manual code to access objects that are instantiated within the generated code.

1.3 Approach

The work in this dissertation addresses the problem of framework usage from two perspectives. On the one hand, it addresses the complexity of framework reuse interfaces, while on the other hand, it addresses the development of F-DSLs. This work can thus be divided into the following approaches:

1. The difficulty of using a framework directly is addressed with a novel approach for building higher level framework reuse interfaces. Such reuse interfaces are composed of *framework specialization aspects*. A specialization aspect is a reusable module that is used in an application to implement the instantiation of a concept provided by the framework.

2. The difficulty of developing F-DSLs is addressed with an approach based on (1), where the specialization aspects are used to completely encode an F-DSL. Such reuse interfaces are called *domain-oriented reuse interfaces* (DORIs) and used to automate the construction of F-DSLs.

Approach (1) is independent from approach (2), whereas approach (2) relies on (1) and cannot be applied independently. These approaches were validated by developing specialization aspects and DORIs for two case study frameworks: JHotDraw (SourceForge, 2006) and Eclipse Rich Client Platform (RCP) (McAffer

& Lemieux, 2005). The proposed constructs for both specialization aspects and DORIs were validated by applying the Engineering Method (Zelkowitz & Wallace, 1998). The proposed constructs were tested on the two frameworks until no further improvement could be achieved and no new constructs were found useful.

The proposed approaches are evolutionary, given that they can be applied to an existing framework "as is". The approaches were experimented on both case studies without modifying the frameworks.

1.3.1 Framework specialization aspects

The development of framework specialization aspects relies on *aspect-oriented programming* (AOP) (Kiczales *et al.*, 1997). AOP provides more powerful localization and encapsulation mechanisms than conventional object-oriented programming languages, which are exploited to modularize framework *hot spots* (Pree, 1995). Each hot spot of the framework is expressed in a specialization aspect, which is an abstract module that is instantiated by *application aspects*. The application aspects represent hot spot adaptations and are the building blocks of a framework-based application. The main role of AOP technology in the context of specialization aspects is related to the capability of untangling, namely the untangling of hot spot adaptations.

The framework that is extended with a set of specialization aspects is referred to as the *base framework*. Specialization aspects are capable of forming a reuse interface that enables the development of framework-based applications at a higher abstraction level than if using a conventional reuse interface (see Figure 1.3). The several concepts provided by the framework are represented in the specialization aspects, while an application is composed by several application aspects that instantiate the specialization aspects. For the same concept instances of a framework-based application, the gap between the concept instances and the application aspects that implement them is shorter than the gap between the concept instances and the application code that conventionally instantiates the framework.

In order to help on the task of enhancing a conventional framework with specialization aspects, this dissertation also presents a pattern language. The several

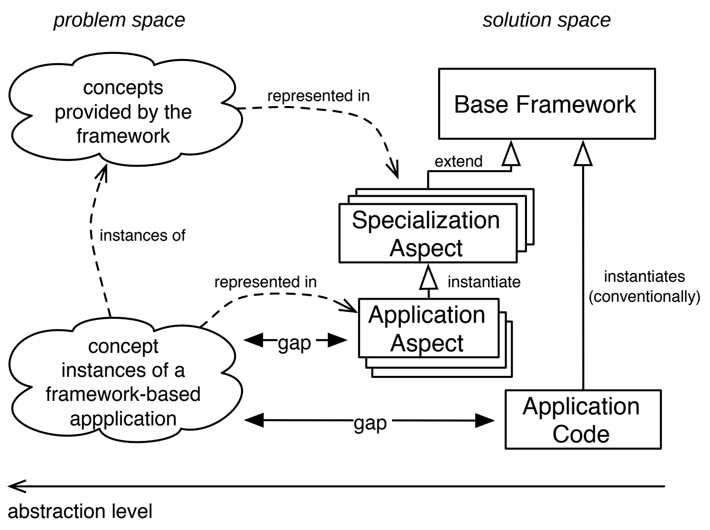


Figure 1.3: Specialization aspects enable to shorten the gap between concepts and framework-based code.

patterns help on finding appropriate situations where specialization aspects are useful and on how to implement them.

1.3.2 Domain-oriented reuse interfaces (DORIs)

The higher abstraction level of specialization aspects is advantageous in the context of F-DSLs, given that the described gap shortening simplifies the transformation of application models into application code. This transformation is straightforward, and therefore, easy to implement in a code generator. At the heart of the transformation simplification is the fact that specialization aspects enable a one-to-one mapping between elements of an application model and application aspects, and thus, they eliminate the scattering and tangling phenomena of model-to-code transformations described in the previous section (see Figure 1.4).

Current strategies for building an F-DSL rely on the development of artifacts that are external to the framework, which define DSL concepts and map them to framework elements. The work in this dissertation proposes a novel technique for building F-DSLs, which is based on having a framework's reuse interface

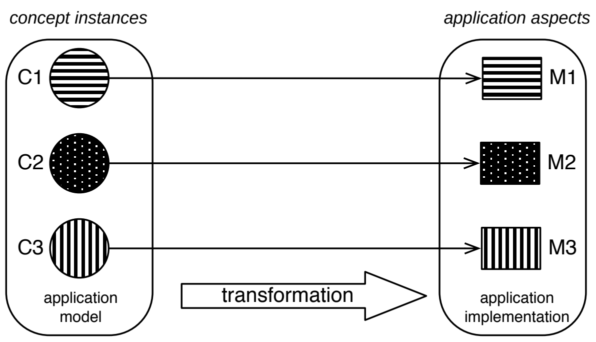


Figure 1.4: Absence of scattering and tangling in model-to-code transformations when using specialization aspects.

closely tied to domain concepts — a *domain-oriented reuse interface* (DORI). Such a reuse interface is structured using specialization aspects, according to the domain concepts and relationships that are represented in a domain variability model. A DORI is a framework layer that encodes an F-DSL.

The way concepts are instantiated using the specialization aspects of a DORI is uniform, in the sense that the instantiation of every concept is achieved using the same mechanism. This enables the definition of a generic transformation that is common to all DORIs. Therefore, by having a framework with a DORI, there is no need of developing a code generator for its F-DSL, given that a generic generator can handle the transformation. DORIs are intended to be used together with a generic *language workbench* (Fowler, 2008), which automatically extracts DSL definitions from a DORI and is capable of transforming application models described in that DSL into DORI-based code. In this way, the development of an F-DSL only relies on a DORI and there is no need to develop any additional artifact.

Figure 1.5 presents an overview of the proposed approach. Framework developers (or product-line engineers) build a DORI for a base framework. Building the DORI requires a combination of framework usage knowledge with a domain variability model that describes the intended DSL concepts. A DSL definition is automatically extracted from the DORI. Application developers describe applications in a model expressed in the DSL according to the application requirements. The model is given as input to the generic code generator in order to generate

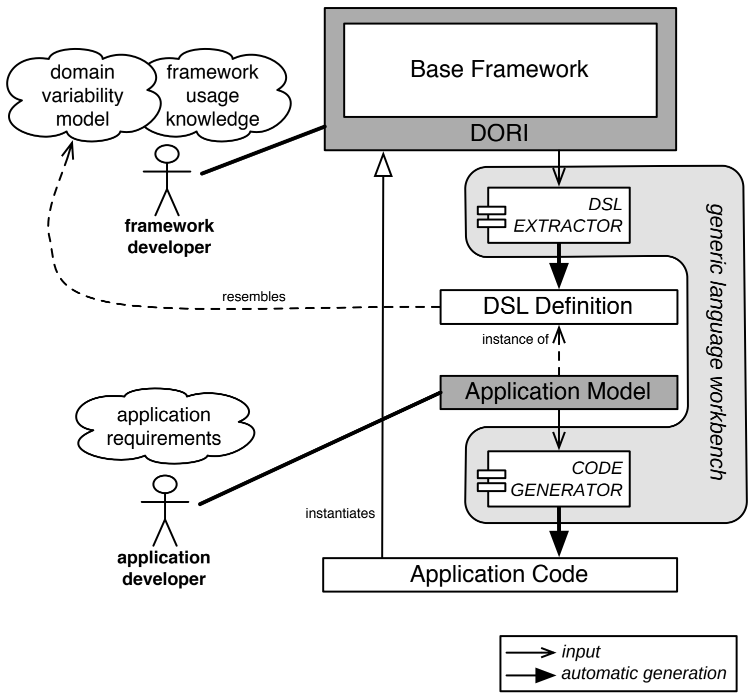


Figure 1.5: DORI-based DSL engineering.

application code, which is based on the DORI rather than directly on the base framework.

The proposed language workbench is intended to be generic, i.e. it can be used for any framework with a DORI implemented in the supported AOP language. This dissertation demonstrates how a language workbench for building *domain-specific modeling languages* (DSMLs) (DSM Forum, 2007) can be implemented. A prototype tool that realizes such a language workbench is presented, which supports AspectJ (Eclipse Foundation, 2007a) as the AOP language and uses EMF (Eclipse Foundation, 2007d) for expressing the DSLs. The tool is named ALFAMA[1] and it is implemented as a set of Eclipse plugins (Eclipse Foundation, 2007b). ALFAMA stands as a proof-of-concept that the proposed approach for automating the construction of F-DSLs is feasible.

With respect to the domain-oriented and implementation-oriented approaches for defining DSL concepts introduced in the previous section, the proposed ap-

[1]Automatic DSLs for using Frameworks by combining Aspect-oriented and Meta-modeling Approaches

proach is somewhat a "unification" of the two, concentrating on the ability to easily implement DSLs with concepts that are purely domain-related. Since the DSL definition is obtained from the DORI, which consists of implementation artifacts, one could consider the definition of the DSL to be implementation-oriented. However, the gap shortening is due to an abstraction level rise in the solution space (the framework), while the definition of concepts sticks to a domain-oriented approach. The idea is to have the benefits of the implementation-oriented approach regarding the simplicity of transformation, while preserving the benefits of the domain-oriented approach regarding the nature of the concepts.

1.4 Contributions

The main contributions of the work in this dissertation are the following:

1. The concept of framework specialization aspects (Santos & Koskimies, 2006; Santos *et al.*, 2006, 2007);

2. A pattern language for designing and implementing framework specialization aspects (Santos & Koskimies, 2008);

3. The concept of DORI, and an effective technique for automating the construction of DORI-based DSMLs (Santos *et al.*, 2008);

4. The implementation of the ALFAMA prototype, a language workbench for building DORI-based DSMLs (Santos, 2007, 2008);

The contributions (1), (3), and (4) were validated through two case studies. The contribution (2) resulted from the experience of applying (1) in the two case studies.

1.5 Thesis Outline

The remainder of this dissertation includes the following chapters:

- Chapter 2 introduces the difficulty of framework learning and presents the JHotDraw framework as a case study. Different strategies for supporting framework usage are presented, going into more detail on the development of F-DSLs, using JHotDraw as an example framework.
- Chapter 3 describes the concept of framework specialization aspects. It is explained how specialization aspects can capture framework hot spots, presenting examples of specialization aspects for the JHotDraw framework.
- Chapter 4 presents a pattern language for designing and implementing framework specialization aspects. A running example of a conventional framework is enhanced with specialization aspects, and the solutions driven by the patterns are compared with the conventional instantiation of the framework.
- Chapter 5 describes the concept of DORI. The constructs that can be used for defining F-DSLs are explained, as well as how they can be represented using specialization aspects.
- Chapter 6 explains how a language workbench for building DORI-based DSMLs can be implemented. The ALFAMA tool is presented as a realization of such a language workbench.
- Chapter 7 presents details about the case studies on JHotDraw and Eclipse RCP. For both frameworks, examples of instances of the patterns of Chapter 4 are given. For a DSML covering a subset of Eclipse RCP concepts, a solution based on having a conventional generator is compared with a DORI-based solution.
- Chapter 8 compares the work in this dissertation with related work.
- Chapter 9 draws the conclusions of the research presented in this dissertation, and outlines future work.

Chapter 2

The Problem of Supporting Framework Usage

This chapter introduces *object-oriented frameworks* and the notion of *hot-spots* in Section 2.1. Section 2.2 presents JHotDraw as a case study framework, explaining how its hot-spots can be adapted and the associated difficulties. Section 2.3 outlines framework usage support strategies. Section 2.4 explains the main tasks in the development of a *domain-specific language* for framework instantiation (F-DSL), using JHotDraw as an example framework. Finally, Section 2.5 outlines the direction that the work in this dissertation follows.

2.1 Object-Oriented Frameworks

An object-oriented framework is a set of classes that embodies an abstract design for solutions to a family of related problems (Johnson & Foote, 1988). A framework is a reusable design together with an implementation, which can be used to build several applications. A framework typically has an architectural nature, given that it defines the overall application structure, the component types and their collaborations, and it assumes the main thread of control.

Frameworks have been used extensively for many years, and they can be classified in three general categories (Fayad *et al.*, 1999):

- *System infrastructure frameworks.* These support the development of system infrastructure in areas ranging from operating systems to graphical user interfaces.

- *Middleware integration frameworks.* These support the integration of distributed applications and components.

- *Enterprise application frameworks.* These address large application domains and support the development of end-user applications directly. This kind of framework is related with software product-lines.

The work in this dissertation is not specially targeted at any specific kind of framework. However, the proposed approaches are likely to better suit enterprise application frameworks. The reason is that, unlike system infrastructure and middleware integration frameworks, enterprise application frameworks typically offer most of the functionality that is necessary to build an application. Therefore, the F-DSL can embrace enough concepts for generating complete applications. Otherwise, the F-DSLs can only cover specific parts of the applications.

2.1.1 Hot spots

A framework provides a set of (domain-specific) concepts for developing applications, which instantiate those concepts according to certain requirements. A framework *hot spot* (Pree, 1995) is a place in the framework where adaptation occurs. Hot spots have to be anticipated by framework developers when designing the framework. Application developers write framework-based code that *adapts* the framework hot spots according to their needs. The adaptation of a hot spot represents the instantiation of a concept provided by the framework.

The set of classes involved in the hot spots of a framework forms its *reuse interface*. Although the number of classes involved in the reuse interface may be large, it is typically a relatively small fragment of all framework classes. Given that a framework's reuse interface is mixed with the remaining framework classes, and the fact that the number of classes of a non-trivial framework is typically large, hot spots can be difficult to locate if they are not documented properly.

Once knowing which are the concepts provided by the framework, the application developer has to locate the corresponding hot spots and understand how to adapt them. The two basic type of elements involved in hot spots are *template methods* and *hook methods*. Template methods define abstract behavior that is completed by hook methods, which are given by framework-based applications for customizing behavior. Therefore, in order to instantiate concepts, application developers have to know which are the relevant hook methods and how to implement them.

Given a certain framework, the kind of hot spot adaptations that are required for building applications characterizes the framework reuse interface:

- *White-box.* This kind of adaptation relies on inheritance and dynamic binding. Framework-based applications have to subclass framework classes and implement hook methods. Although inheritance is a powerful mechanism, it usually requires understanding framework internals, at least to a certain degree of detail.

- *Black-box.* This kind of adaptation relies on object composition and delegation. In this case, application developers only need to care about the external interfaces of the framework classes.

Black-box frameworks are usually a result of several development iterations and are harder to design. On the other hand, they are less flexible than white-box frameworks. Frameworks typically have a *gray-box* reuse interface, implying that some of its hot spots have to be adapted with white-box mechanisms, while others require black-box mechanisms.

Hot spots may rely on different object-oriented mechanisms or combinations of them. The fundamental mechanisms that are supported by all the mainstream object-oriented languages (such as Java, C++, or C#), are the following:

1. *Class inheritance.* An application class inherits from a framework class.

2. *Hook method overriding.* An application class overrides hook methods of the classes obtained in (1) for customizing behavior.

3. *Object composition.* Instances of framework classes are composed in order to include components in an application or to establish associations between them.

4. *Object parameterization.* Instances of framework classes are parameterized in order to customize the components according to application requirements.

5. *Interface implementation.* An application provides a component by implementing a framework interface.

Mechanisms (1) and (2) are the fundamental mechanisms for white-box adaptation, whereas mechanisms (3) and (4) are the fundamental mechanisms for black-box adaptation. Typically, the adaptation of a certain hot spot involves a combination of two or more mechanisms. For instance, the combinations of (1) and (2), (3) and (4), (2) and (3), (3) and (5), occur often.

A hot spot may involve several classes. On the other hand, typically only fragments of those classes (e.g. some of their methods) are relevant in the context of a particular hot spot, and common fragments may be related to different hot spots. Given that a hot spot is associated with a concept provided by the framework, this implies a many-to-many mapping between concepts and classes. Therefore, the instantiation of a single concept may involve one or more framework classes, while one framework class may be relevant for instantiating zero or more concepts. Figure 2.1 illustrates this phenomena.

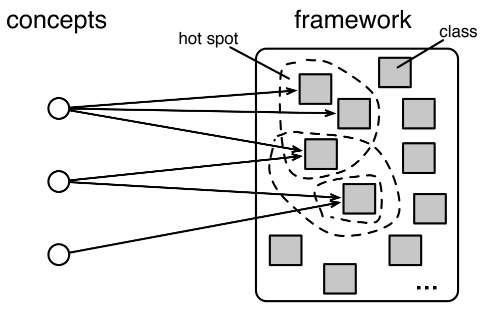


Figure 2.1: Mapping between framework concepts and hot spots' classes.

2.2 Case Study: JHotDraw

JHotDraw serves the purpose of building *drawing applications* for structured graphics. Figure 2.2 presents a screenshot of a JHotDraw-based application, annotated with the main framework concepts. A drawing application has a tool bar that contains *creation tools* (which can optionally have *undo* support) for inserting *node figures* (e.g. *rectangle*, *ellipse*) and *connection figures* (e.g. *line*, *arrow*) on the drawings. For a given connection figure, valid source and target node figures can be specified. For example, the line can only connect rectangles to ellipses. The application can have *menus*, which contain *commands*. The framework provides a set of default figures and connection figures, as well as default commands (e.g. *exit*, *save*). On the other hand, JHotDraw-based applications may include application-specific figures, connection figures, or commands, by implementing framework interfaces.

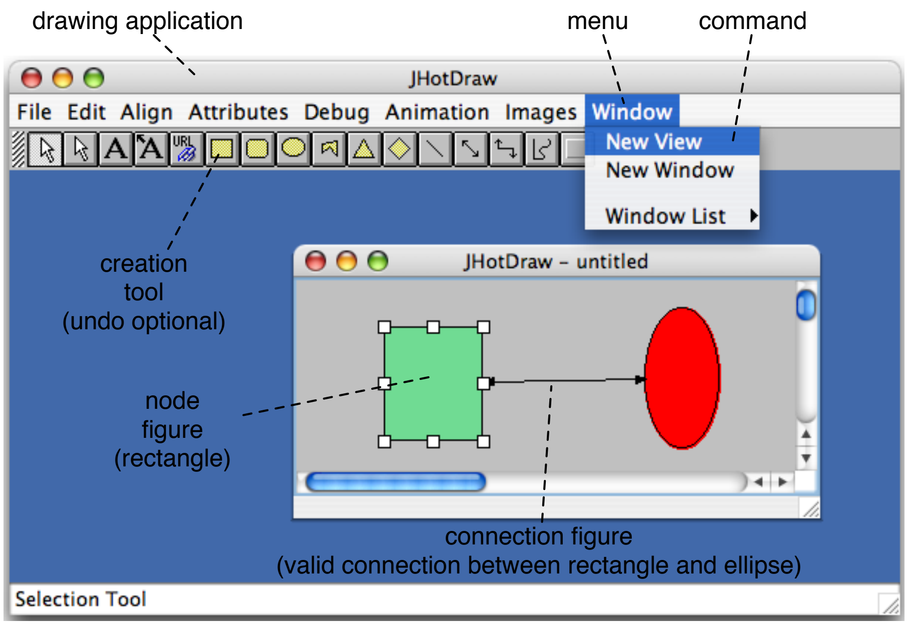

drawing application
menu
command
JHotDraw
File Edit Align Attributes Debug Animation Images Window
New View
New Window
Window List
creation tool (undo optional)
JHotDraw – untitled
node figure (rectangle)
connection figure (valid connection between rectangle and ellipse)
Selection Tool

Figure 2.2: Screenshot of an application based on JHotDraw.

2.2.1 Difficulty of understanding hot spots

One of the main problems of framework-based development is the steep learning curve (Moser & Nierstrasz, 1996; Pree, 1995). Application developers have to know which are the concepts provided by the framework, and then to locate the hot spots associated with those concepts within the large set of framework classes. Given the many-to-many mapping between concepts and hot spots' classes (depicted in Figure 2.1), it is difficult to understand a hot spot in isolation if its classes are also involved in other hot spots. This issue is related with the *scattering* and *tangling* phenomena.

In the general context of translating requirements to implementation, the scattering and tangling phenomena compromise software comprehension and evolution (Tarr *et al.*, 2004). On the one hand, the implementation of framework-based applications suffers from scattering due to the fact that a single requirement — associated with an instance of a framework concept — may involve more than one application class. On the other hand, framework-based applications also suffer from tangling given that a single application class may be related to more than one requirement.

Table 2.1 presents a list of concepts of a JHotDraw-based application, and the adaptation mechanisms that are involved in each concept usage. The table contains the essential concepts that are used in almost every application. Notice that there is at least one case for each adaptation mechanism, as well as several examples of combination of adaptation mechanisms.

Figure 2.3 presents the fragment of JHotDraw's reuse interface that is relevant for the concepts listed in Table 2.1, while the following list details how an application developer has to proceed in order to use those same concepts. Each concept is identified by the *id* number given in the table. When the concept requires developing a new class, a letter is used to identify that class.

1. Implementing an "empty" drawing application requires developing a class (A) that inherits from DrawApplication, which has the hook method createTools(JToolBar) for plugging figures and creation tools, and the hook method createMenus(JMenuBar) for plugging menus and commands on those menus.

Id	Concept	Inheritance	Interface	Hook Method	Object Composition	Parameters
1	Draw application (empty)	√				
2	Default node figure (e.g. Rectangle)					√
3	Application-specific node figure		√			
4	Default connection figure (e.g. Line)					√
5	Application-specific connection figure		√			
6	Valid connection on (4)	√		√		
7	Valid connection on (5)		√			
8	Creation tool for (2-5)			√	√	
9	Undo on (8)				√	
10	Menu on (1)			√	√	√
11	Default command on (10)				√	
12	Application-specific command on (10)		√		√	

Table 2.1: Concepts of a JHotDraw-based application.

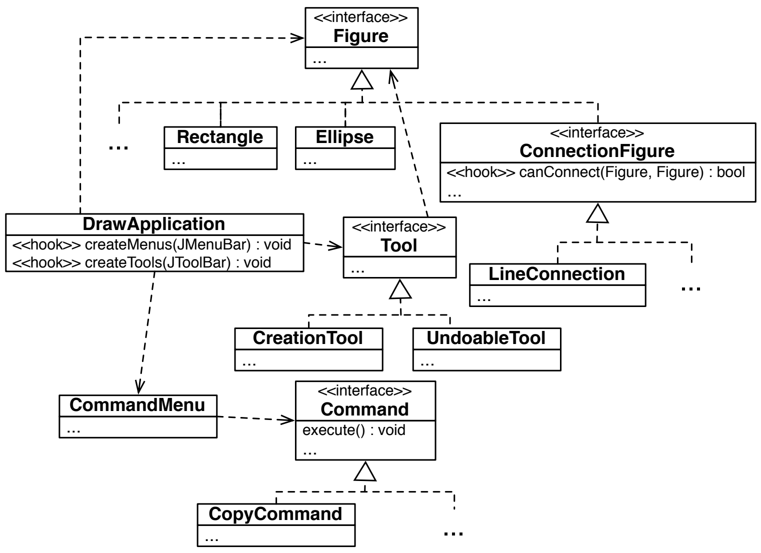


Figure 2.3: Fragment of JHotDraw's reuse interface.

2. Using a default node figure requires instantiating a framework class that implements the interface Figure, as for instance Rectangle or Ellipse.

3. Using an application-specific node figure entails developing a class (*N*) that implements the interface Figure.

4. Using a default connection entails instantiating a framework class that implements the interface ConnectionFigure, as for instance LineConnection. The interface ConnectionFigure has the hook method canConnect(Figure, Figure) for customizing the valid source and target node figures that the connection figure may connect.

5. Using an application-specific connection figure entails developing a class (*L*) that implements the interface ConnectionFigure.

6. Defining a valid connection for a default connection figure entails developing a class (*V*) that inherits from the framework class that implements the interface ConnectionFigure, overriding the hook method canConnect(Figure, Figure).

7. Defining a valid connection for an application-specific connection figure entails modifying the class *L* introduced in (5) (which implements the connection figure), namely on the hook method canConnect(Figure, Figure).

8. Including a creation tool entails modifying the class *A* introduced in (1), by overriding the hook method createTools(JToolBar). Figure or connection figure objects (2-5) have to be wrapped in an instance of the framework class CreationTool and plugged in the parameter of type JToolBar.

9. Including undo support on a creation tool entails modifying the class *A* introduced in (1), on the hook method createTools(JToolBar). The CreationTool objects introduced by (8) have to be wrapped in an instance of UndoableTool.

10. Including a menu in the application entails modifying the class *A* introduced in (1), by overriding hook method createMenus(JMenuBar). One has to plug an instance of CommandMenu in the parameter of type JMenuBar.

11. Including a default command in a menu entails modifying the class *A* introduced in (1), namely the hook method createMenus(JMenuBar). One has to plug an instance of a framework class that implements the interface Command (e.g. CopyCommand in the menu objects introduced in (10)).

12. Including an application-specific command entails developing a class (*C*) that implements the interface Command. Such a command can be plugged as in (11).

Figure 2.4 depicts the relations between the tasks associated with the concepts' usage, namely regarding (a) development of a new class in the application, (b) modifications of the classes of an application, and (c) tangled instantiation of concepts. Consider the tangling relations to be transitive, i.e. if *a* is tangled with *b*, and *b* is tangled with *c*, then *a* is tangled with *c*.

Given the difficulties related to learning, a framework can hardly be given to application developers without some form of usage support, which relieves the burden of having to learn the framework just by reading its classes. In order to support framework usage, several approaches following different strategies have been proposed, which are detailed next.

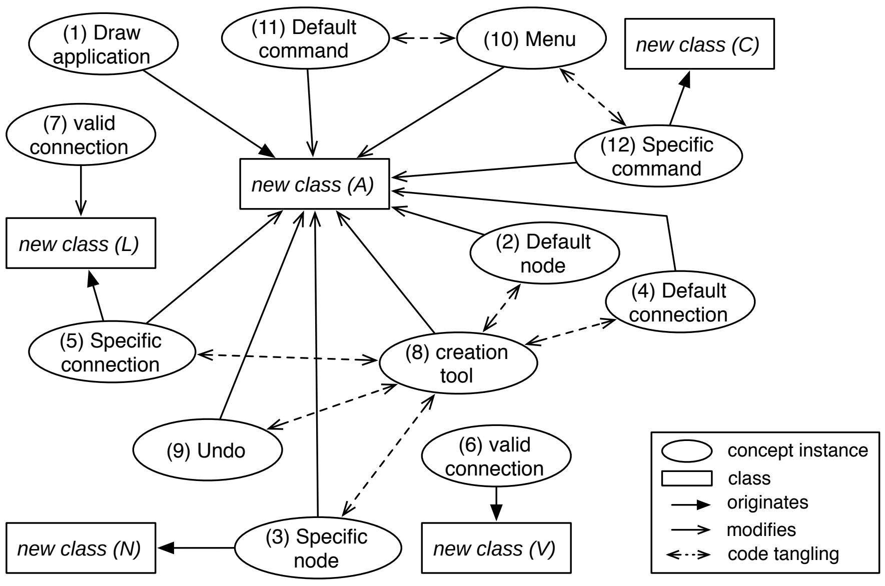


Figure 2.4: Conventional JHotDraw instantiation.

2.3 Framework usage support strategies

Due to the reasons explained previously, it is extremely important a framework to be accompanied with some sort of support for using it. The usage support can follow one of three main strategies: *documentation*, *instantiation tools*, and *domain-specific language* (referred to as F-DSL in this dissertation).

2.3.1 Documentation

This kind of support consists in providing manuals for learning how to use the framework. A manual can be organized in different ways. A popular way is to have documentation in the form of a *cookbook* (Krasner & Pope, 1988), where several "recipes" for adapting the framework hot spots are presented to application developers, typically using examples. Another form of organizing framework documentation is by means of design patterns (Johnson, 1992), where several framework-specific patterns are organized in a collection or in a pattern language. Although each pattern has the role of a "recipe", documentation organized in patterns is more structured.

Steyaert *et al.* (1996) proposed the concept of *reuse contracts*, which are interface descriptions that offer guidelines for reusing assets (e.g. frameworks). Although one of the main purposes of reuse contracts is to aid on managing evolution of reusable assets, they can be used as structured documentation of a framework's reuse interface that assists application developers in the development of framework-based applications.

The cost of producing good framework documentation is typically high (Fayad *et al.*, 1999). In order to reduce this cost, an approach for developing minimalist documentation has been proposed by Aguiar (2003), having in mind also the purpose of facilitating the use of the documentation by application developers.

Documentation approaches just provide additional information for framework learning, while they do not hide framework complexity. Moreover, in addition to the cost of producing the documentation, there is also the cost of maintaining it. Case studies on industrial settings report that existing framework documentation is often either inconsistent or incomplete (Bosch, 1999). In order to mitigate the lack of documentation, it is possible to mine usage examples from code repositories (e.g. Holmes *et al.*, 2005). However, this requires a rich repository of framework-based code, and it also does not hide framework complexity.

2.3.2 Instantiation tools

Instantiation tools support framework usage by assisting the development of framework-based applications. These tools are based on formalization of framework-specific patterns, which are used to guide the framework instantiation process and to automate certain tasks (e.g. partial code generation). SmartBooks (Ortigosa *et al.*, 2000), JavaFrames (Hakala *et al.*, 2001), and Design Fragments (Fairbanks *et al.*, 2006), are examples of such tools. Instantiation tools require framework developers to annotate the framework, either internally or externally. Hautamäki & Koskimies (2006) propose an approach to find, specify and use the reuse interface of a framework by means of framework-specific patterns.

Instantiation tools are an enhanced form of framework documentation, which is capable of automating certain tasks. Although the complexity of the framework is partially hidden from application developers, framework-based applications are given in terms of solution space abstractions. Application developers have to manipulate framework-based code, and therefore, they need to understand the reuse interface up to a certain extent.

Viljamaa (2003) proposes an approach to reverse engineer framework reuse interfaces. However, such an approach is not completely reliable, especially in terms of completeness and precision.

2.3.3 DSL

The strategy that is commonly agreed to be the most effective for supporting framework usage is to develop an F-DSL, which is used for generating framework-based code from high-level application descriptions. By having an F-DSL, the framework may become completely hidden from application developers, and the abstraction level can thus be raised. It might happen that the application developers are not programmers, but rather domain experts, given the abstraction level raise that this strategy allows. This kind of solution is usually realized with the aid of *language workbenches* (Fowler, 2008). Language workbenches are IDEs for suitable for developing DSLs. MetaEdit+ (MetaCase, 2008), Microsoft DSL Tools (Greenfield & Short., 2005), and GME (Ledeczi *et al.*, 2001), are examples of such language workbenches.

The three strategies for framework usage support differ in terms of realization effort, and effectiveness in terms of the aid they provide to application developers. F-DSL approaches require the biggest up-front effort, but are definitely the most effective strategy. Documentation approaches typically require the least effort to realize, but accordingly, they are likely to be the least effective. Instantiation tools typically require more effort than documentation, and they are roughly in between documentation and F-DSLs with respect to effectiveness.

2.4 DSLs for Instantiating Frameworks

DSLs work well with frameworks given that both have the goal of creating related applications within a certain domain. However, non-trivial F-DSLs are in general not easy to realize. This section explains the necessary tasks for realizing an F-DSL and the associated difficulties.

There are three fundamental tasks involved in the realization of an F-DSL:

1. *Identification of concepts and relationships.* Through extensive domain analysis, the scope of domain concepts and how they relate has to be defined. Although the concepts should have been identified when building the framework in the first place, in many cases this is likely not to have happened.

2. *Language definition.* Once (1) is achieved, the F-DSL syntax has to be formally defined. It is important to distinguish between the abstract and concrete syntax of the DSL. Conceptual models are suitable for expressing the concepts and their relationships. The abstract syntax represents the underlying structure of concepts, which can be expressed in a formal conceptual model. The concrete syntax is what is visualized when the F-DSL is being used.

3. *Transformation definition.* Once having (2), a transformation between F-DSL concepts and framework-based code has to be defined. The transformation can be defined in terms of the abstract syntax of the language.

The next subsections detail these tasks and discuss the problems that are associated to them.

2.4.1 Identification of concepts and relationships

A framework provides a set of concepts that can be instantiated in a framework-based application. The concepts have relationships between them. Often, framework concepts are not well-defined and their scope is not delimited. However, in order to develop an F-DSL, the concepts that are to be covered by the language have to be clearly identified, given that they are going to be first-class entities in the F-DSL. The concepts have to be identified with a certain degree of precision, so that they can be transposed to a formal language definition seamlessly.

Especially when a domain is not stable, it might be hard to set the scope of the concepts definition, given that new concepts emerge frequently. This implies constant modifications in the F-DSL (syntax and transformation). If the points where new concepts can be introduced were not anticipated, the cost of evolving the F-DSL constantly is likely to be high. In order to avoid frequent F-DSL modifications, the solution may take into account mechanisms for integrating manual code with code that is generated from the models, so that features that are not represented in the F-DSL can be manually encoded. Such mechanisms may be referred to as *escapes* (Cleaveland, 1988).

The work in this dissertation does not address domain analysis in any way. It is assumed that domain analysis has been performed, so that the framework concepts for developing an F-DSL have been identified. For instance, assume that the concepts of JHowDraw described previously would set the scope of an F-DSL. Such concepts will be used in the following sections for presenting a running example.

2.4.2 Language definition

Using model-driven engineering terminology, DSL concepts and their relationships can be defined in a *meta-model*, while a description in a certain DSL is a *model*, which is an instance of the meta-model. Figure 2.5 presents a conceptual model for applications based on the JHowDraw framework, describing the concepts, their attributes, and the relationships between them. Such a conceptual model can be thought of as being a meta-model that defines the abstract syntax of the F-DSL. For instance, the meta-model can be formally defined using EMF (Eclipse Modeling Framework, Eclipse Foundation, 2007d), which is a meta-modeling technology where meta-models are given as conceptual models.

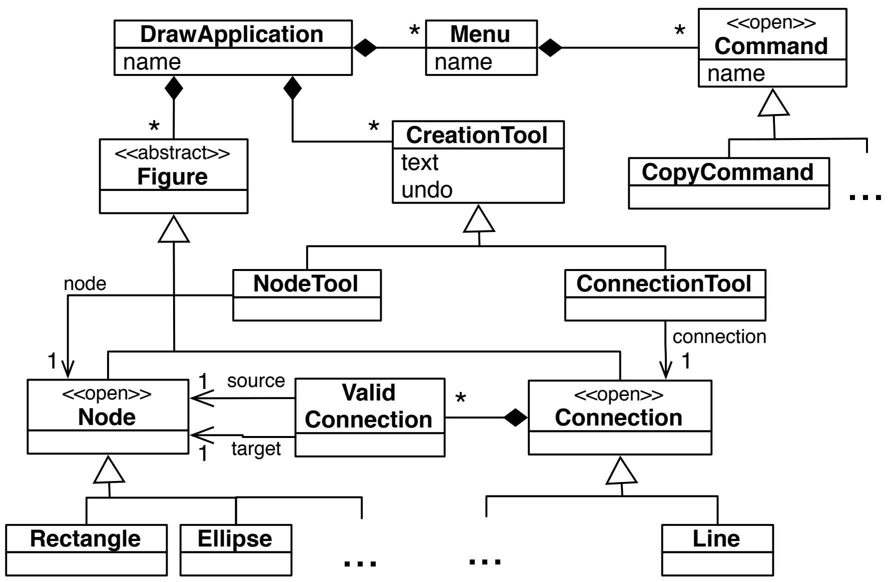
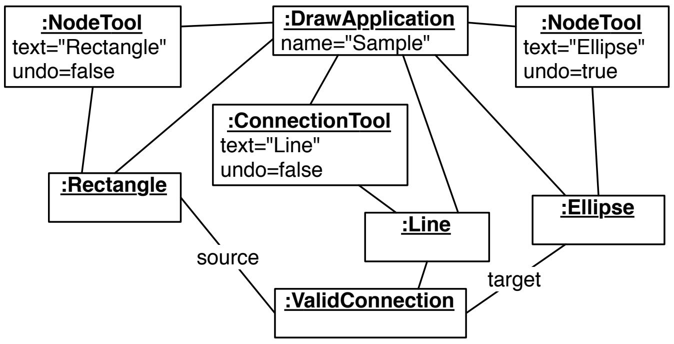
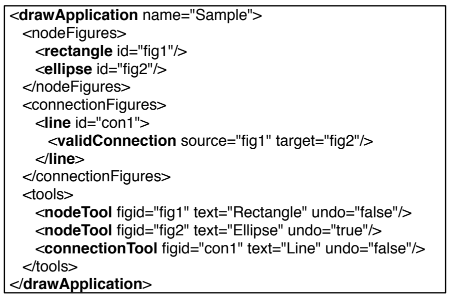


Figure 2.5: Domain variability model for JHotDraw (simplified).

In Figure 2.6(a) we can see an object diagram describing an application model (i.e. instance of the model of Figure 2.5). The object diagram is an abstract representation of the application model expressed in the F-DSL. It describes an application named “Sample” that makes use of two node figures, rectangle and ellipse, and the line connection figure. Each figure has a creation tool for it, and the creation tool for the ellipse figure has undo support.

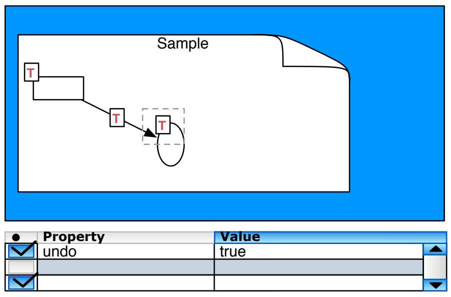


(a) Example application model



```
<drawApplication name="Sample">
  <nodeFigures>
    <rectangle id="fig1"/>
    <ellipse id="fig2"/>
  </nodeFigures>
  <connectionFigures>
    <line id="con1">
      <validConnection source="fig1" target="fig2"/>
    </line>
  </connectionFigures>
  <tools>
    <nodeTool figid="fig1" text="Rectangle" undo="false"/>
    <nodeTool figid="fig2" text="Ellipse" undo="true"/>
    <connectionTool figid="con1" text="Line" undo="false"/>
  </tools>
</drawApplication>
```

(b) Application model of (a) represented with XML



Property	Value
undo	true

(c) Application model of (a) represented with a dedicated GUI

Figure 2.6: Different concrete syntax for a same application model.

A F-DSL may have multiple concrete syntaxes. In Figure 2.6 we can see a same application model represented using two hypothetical representations for the concrete syntax. In Figure 2.6(b) we have a concrete syntax based on XML, while in Figure 2.6(c) we have a graphical editor for manipulating the application models. The definition of a concrete syntax for an F-DSL is a relevant issue, especially concerning usability. However, the development of concrete syntax is an orthogonal issue to the abstract syntax and transformation that can be addressed independently.

Even if the domain associated with the F-DSL is stable, certain solutions necessarily require the existence of *open variation points* (Gurp *et al.*, 2001). These are points where it is possible to introduce a variant that cannot be predefined. For instance, given that the behavior of a JHotDraw command can be almost anything, it is not reasonable to consider the development of an F-DSL capable of modeling the behavior of any command. In this case, given that there are infinite commands that one may define, the variability with respect to a command's behavior cannot be anticipated, and therefore, *command* should be considered an open variation point.

2.4.3 Transformation definition

Application models described in an F-DSL are transformed to a program written in a *target* language. In the context of this dissertation, the target language is the general purpose object-oriented language in which the framework-based code is written. The transformation definition is likely to be the most difficult task in DSL engineering. The problem is that the gap between the F-DSL concepts and framework-based code may lead to complex transformations.

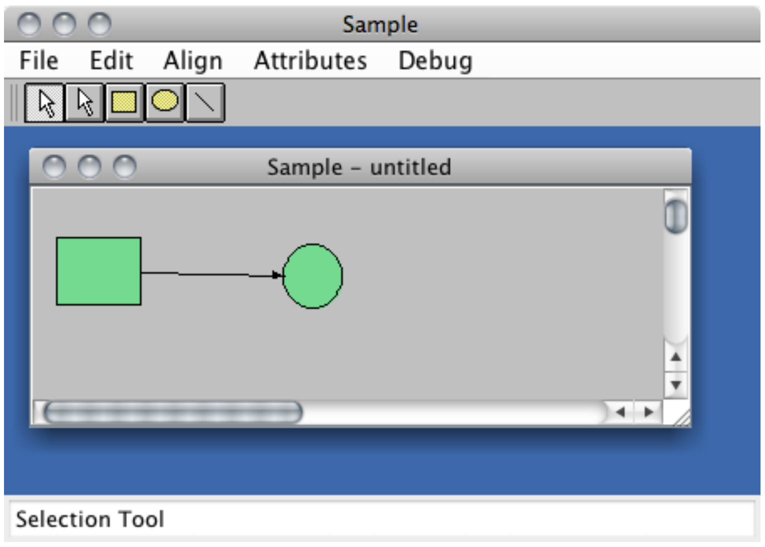
Figure 2.7 presents the code of the JHotDraw-based application described by the the application model given in Figure 2.6. The given code is what a code generator has to output. According to the content of Figure 2.4, notice the tangling of code pertaining to figures, tools, and undo support.

In principle, the bigger the gap between concepts and framework-based code is, the more complex the transformation definition. The concepts/code gap varies according to the framework reuse interface. White-box reuse interfaces usually have bigger gaps (e.g. due to hook methods), and therefore they imply more complex transformations. Black-box reuse interfaces usually have a shorter gaps, given that the framework-based code consists in object instantiation and composition, resembling concept instances and their relationships more closely. It has been pointed out by Roberts & Johnson (1997) that black-box frameworks have an adequate maturity for having an accompanying F-DSL. However, it is important to take into account that black-box frameworks are harder to implement, and that many frameworks never reach this stage.

Although the concepts/code gap may be shorter in black-box frameworks, both the concepts and their mapping to framework-based code is implicit. Therefore, the definition of both the concepts and the transformation has to be explicitly given elsewhere. The definition of concepts was addressed previously. Concerning the transformation definition, it is typically implemented in a code generator. Despite the particular technology that is adopted for this purpose,

```
public class SampleApplication extends MDI_DrawApplication {

 public SampleApplication() {
  super("Sample");
 }

 public void createTools(JToolBar bar) {
  super.createTools(bar);

  Tool tool = new CreationTool(this,new RectangleFigure());
  bar.add(createToolButton("RECT", "Default Rectangle",tool));

  tool = new UndoableTool(new CreationTool(this,new EllipseFigure()));
  bar.add(createToolButton("ELLIPSE", "Default Ellipse",tool));

  tool = new ConnectionTool(this,new CustomLineConnection());
  bar.add(createToolButton("LINE", "Default Line",tool));
 }
}
```

```
public class CustomLineConnection extends LineConnection {
 public boolean canConnect(Figure src, Figure tgt) {
  return
    src instanceof RectangleFigure &&
    tgt instanceof EllipseFigure;
 }
}
```



Selection Tool

Figure 2.7: JHotDraw-based code that implements the model of Figure 2.6, and corresponding screenshot.

the role of the code generator is to explicitly map concepts to framework-based code. As explained, depending on the reuse interface of a given framework, the complexity of the mapping may vary. The more complex the mapping is, the more difficult the implementation and maintenance of the transformation in a code generator.

2.4.3.1 Difficulties of generating framework-based code

Although some elements of a reuse interface are likely to have a close relation with the elements of a domain variability model, the reuse interface can be complex enough so that the transformation of concepts to code is non-trivial. This is so due to the following reasons:

- *Non-uniform representation of concepts in the reuse interface.* Different domain concepts can be used in an application using different means, such as subclassing, object parameters, or object composition. This implies that an application developer has to find out how to instantiate a certain domain concept by inspecting an heterogeneous reuse interface, in the sense that the mechanisms used to instantiate the concepts differ from case to case.

- *Manipulation of domain-unrelated elements.* Application code may need to include elements that have no correspondence to a domain concept, for instance the definition of a hook method. Such domain-unrelated elements introduce accidental complexity in the task of finding out how to instantiate the concepts.

- *Tangled application classes.* The code of a framework-based class typically includes statements pertaining to different concept instances. The generation of interleaved code that originates from the different concept instances represented in an application model is an additional burden that has to be managed.

- *Code dependencies.* It is typically necessary to have dependencies between code generated from different concept instances. This issue may cause additional complexity in the code generator implementation because the generator parts become interdependent, for instance due to the need of managing variable names that have to be shared.

Besides these difficulties related with the nature of framework-based code, other more general difficulties hinder the development of code generators:

- *Understandability.* Program generation is hard to understand, given that one has to reason about the generated program indirectly through another program — i.e. the code generator. This indirection can be a burden for the one that programs the code generator.

- *Consistency and traceability.* Code generators produce text that constitutes a program. Therefore, the generator implementation deals with program statements that contain identifiers of elements of the reuse interface. These identifiers are manipulated as strings, which are treated in a non-compilable way — e.g. a misspelled keyword or method name on these statements does not indicate a compile error. Therefore, code generators are fragile. A modification in the reuse interface may cause an unnoticeable error in the code generator — the code that is being produced becomes incorrect, while the framework developer does not get any associated error or warning.

- *Integration of manual code.* Ideally, it is not intended that the application developer has to manipulate or even understand code that is generated. Therefore, a proper mechanism to integrate manual code should not require the application developer to perform any inspection or manipulation of generated code. One of the difficulties of realizing such mechanisms is to enable manual application code to access objects that are instantiated within the generated code.

2.5 Towards the Proposed Approach

This chapter presented an overview of existing framework usage support strategies, comparing them and presenting their limitations. The adoption of F-DSLs was explained in more detail given that the work in this dissertation concentrates on improving the way they can be built.

All the framework usage support strategies covered in Subsection 2.3 have an essential characteristic in common. The knowledge of how to instantiate a framework — which is presumably mastered by framework developers — is represented externally to the framework in a dedicated artifact for that purpose (i.e. documentation, instantiation tool, or DSL). In order to facilitate the usage of existing frameworks, none of these approaches proposes changes in the way frameworks themselves are implemented.

As opposed to the described usage support strategies, already more than a decade ago, Johnson (1997) expressed the wish for improved mechanisms in programming languages so that framework-based applications could be expressed with more conceptual clarity. In this way, the framework would become easier to use, and application developers would be less dependent on other artifacts, such as documentation. The approaches that have been proposed since then do not follow this kind of strategy.

The work in this dissertation goes in the direction of raising the abstraction level of reuse interfaces, so that external artifacts lose their relevance, or in the best case, become unnecessary. A higher abstraction level in a reuse interface enables easier framework instantiation. On the other hand, it is advantageous when building F-DSLs to have a framework with a higher level reuse interface. Having a short gap between concepts and code is pointed out by DSL engineering experts as a means to keep code generators simple, based on their experiences with several DSL solutions (Pohjonen & Tolvanen, 2002).

The work in this dissertation also proposes to build such higher level reuse interfaces in order to bridge the gap between concept instances of an application model and framework-based code.

There are two extreme strategies to bridge the gap, given a set of concepts and a framework:

1. *Lower the abstraction level of concepts.* This is referred in Chapter 1 as the *implementation-oriented* approach, where the F-DSL concepts become closer to implementation elements, implying that they are not purely domain-related. This option does not explore the full potential of raising the abstraction level that an F-DSL may offer.

2. *Raise the abstraction level of the reuse interface.* In opposition to strategy (1), the concepts remain at the same abstraction level, but the abstraction level of the reuse interface is raised. This strategy is compatible to what is referred in Chapter 1 as the *domain-oriented* approach, given that it allows F-DSLs to have higher abstraction levels.

It is important to recall that the more domain-oriented the definition of F-DSL concepts is, the higher the abstraction level of the F-DSL. The work proposed in this dissertation enables the gab to be bridged following the direction of strategy (2). Therefore, the approach enables the abstraction level of the reuse interface to become closer to the domain concepts.

Chapter 3

Framework Specialization Aspects

This chapter addresses the concept of *framework specialization aspects* (or simply, specialization aspects), and how they can express framework hot spots. Section 3.1 presents an overview of the approach. Section 3.2 explains concepts of *aspect-oriented programming* (AOP), namely the subset of primitives that are required for developing specialization aspects. Section 3.3 details the concept of specialization aspects. Section 3.4 explains how framework hot spots can be expressed with specialization aspects. Section 3.5 summarizes the characteristics which make specialization aspects a suitable means for developing higher level reuse interfaces. Finally, Section 3.6 discusses some issues regarding the suitability of AOP languages for implementing specialization aspects.

3.1 Overview

A framework provides several concepts that can be used in a framework-based application. Such concepts can be instantiated by adapting the hot spots that are present in the framework's reuse interface.

A specialization aspect (SA) is a building block for a new, higher level, reuse interface that can be built on top of the reuse interface of an existing *base framework*. The proposed reuse interface is composed of several SAs, which express the hot spots of the base framework. The hot spots are implicit in conventional reuse interfaces, and the goal is to make them explicit using SAs. Each SA represents a concept provided by the framework and it is implemented as an abstract module that is part of the new reuse interface. It is intended that SAs can be developed without modifying the base framework.

The main goal of SAs is to allow framework instantiation at a higher abstraction level than conventional instantiation. A higher abstraction level enables the instantiation of domain concepts to be closer to the developer intent. An *application aspect* (AA) is a building block of a framework-based application that adapts one SA by developing a concrete module that inherits from the SA. While the SA represents a concept, an inherited AA represents an instance of that concept. There may exist several AAs inheriting from the same SA, representing thus multiple instances of the concept within the same application. AAs are composed only through static references, i.e. an AA is composed with another AA just by referencing its module. Such static references represent relationships between concept instances. An AA instantiates a concept cohesively, meaning that the concept instance is solely implemented by a single AA, while no modification of other AAs is required. The development of an AA does not require any knowledge about the internals of the other AAs.

The development of SAs is a responsibility of framework developers (or domain engineers), which are supposedly the ones who have better knowledge of how the base framework should be used. An application developer can then select SAs according to the specific requirements of the application, and develop AAs in order to implement a framework-based application. Figure 3.1 depicts SAs and AAs. An SA is a reusable module provided by the framework, whereas an AA is a concrete module that inherits from an SA. The higher level reuse interface is composed of a set of SAs, which form a “wrapper” to the base framework. Given that SAs wrap the base framework, they depend on their classes. Moreover, SAs may have mutual dependencies since they collaborate with each others. In addition, SAs may have inheritance relationships. In this way, it is possible to generalize common behavior among a set of related SAs, avoiding code duplication, and enabling easier extensibility. A framework-based application is composed by several AAs with static dependencies between them.

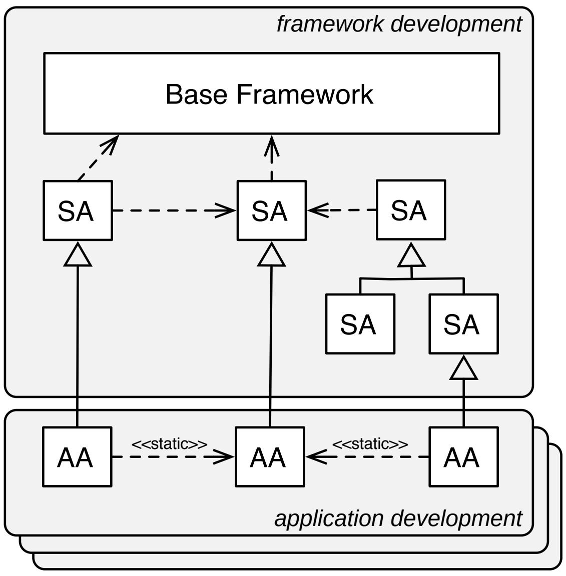


Figure 3.1: Specialization aspects and application aspects.

3.2 Aspect-Oriented Programming

This section presents some basic concepts of AOP (Kiczales *et al.*, 1997) illustrated with the AspectJ language (Eclipse Foundation, 2007a). The AOP features given here are the ones that are necessary to understand Sections 3.3-3.4. A reader familiar with AOP and AspectJ may skip this section.

3.2.1 Paradigm

In the context of object-oriented programming, the use of AOP involves a *base program* composed by classes, and additional modules that are referred to as *aspects*. The purpose of aspects is to modularize what otherwise could not be modularized just by using classes. Aspects are entities that introduce behavior in the base program, ideally in an *oblivious* way (Filman & Friedman, 2004). By obliviousness it is meant that the base program does not need to be aware of the existence of the aspects. The process of combining the aspects with the base program is called *weaving*.

What can be modularized in an aspect is commonly referred to as a *crosscutting concern*, given that the behavior represented in the aspect cuts across the structure of the base program, i.e. the classes. Typical crosscutting concerns are related with system infrastructure, as for instance, security, persistence, logging, or distribution. Crosscutting concerns can be characterized as *homogeneous* or *heterogeneous* (Colyer & Clement, 2004). The former are concerns which affect several classes of the base program in the same way, whereas the latter are concerns which affect several classes in different ways (see Figure 3.2).

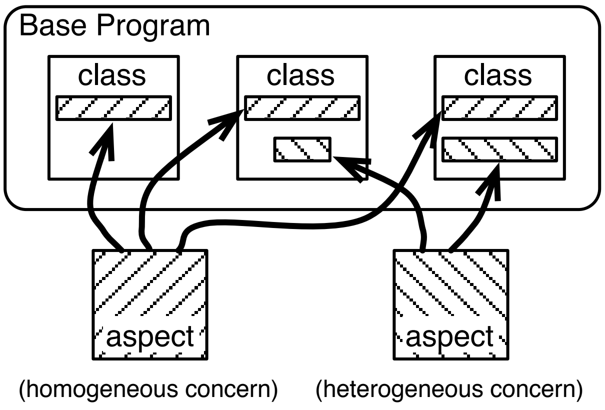


Figure 3.2: Weaving of aspects in a base program: homogeneous and heterogeneous concerns.

3.2.2 AOP concepts

This subsection overviews the main concepts of AOP and exemplifies how they are expressed in the AspectJ language.

Join points

A *join point* is a point in the execution flow of a program that can be unambiguously identified. An aspect is capable of intercepting the execution of the base program at its join points. Moreover, the context of the execution point can be accessed, and behavior can be introduced at those points. There are several kinds of join points. The following list presents the main kinds:

- *Method execution.* An aspect intercepts a method execution. The object where the method is executed, the parameters, and return value, can be accessed and modified by the aspect.

- *Method invocation.* Invocations of a certain method can be in several parts of the program, whereas the method itself is only in one place. This kind of join point differs from the previous one in the sense that the aspect intercepts the execution point of the method call. Besides what can be accessed in a method execution join point, the aspect can additionally access the context where the method was invoked, i.e. the object that made the call.

- *Object instantiation.* An aspect intercepts the creation of objects of a certain type. As well as in the case of regular methods, the constructor can be accessed either when it is executed or invoked.

- *Attribute access.* An aspect intercepts the execution points where the base program is accessing (reading or modifying) an attribute of an object of a certain type.

Pointcuts

A *pointcut* is a specification of a set of join points. Using AspectJ syntax, some examples of pointcut definitions are given in the following list:

- Capturing the execution of a method with signature void someMethod1():

  ```
  pointcut execSomeMethod1() : execution(void someMethod1());
  ```

- Capturing the execution of a method with signature void someMethod2(Type), with access to its parameter:

  ```
  pointcut execSomeMethod2(Type t) :
     execution(void someMethod2(Type)) && args(t);
  ```

  The primitive args plus the declaration of the parameter (Type t) enables the parameter to be accessed through the variable t.

- Capturing the invocation of a method with signature void someMethod1():

```
pointcut invocationSomeMethod1() : call(void someMethod1());
```

- Capturing the instantiation of objects of type SomeClass:

```
pointcut newSomeClass() : initialization(SomeClass.new());
```

- Capturing the read accesses to attribute someAtt of SomeClass:

```
pointcut readSomeAttr() : get(SomeClass.someAttr);
```

- Capturing the write accesses to attribute someAttr of SomeClass:

```
pointcut writeSomeAttr() : set(SomeClass.someAttr);
```

In the case of using *abstract aspects*, pointcuts can be declared as *abstract*, demanding *concrete aspects* that inherit from that aspect to define the pointcuts. An example of a declaration of an abstract pointcut is:

```
abstract pointcut unknownJoinPoints();
```

Pointcuts can be a defined as an intersection of different pointcuts. For instance, a pointcut may be defined on the execution of someMethod1() of the owner class SomeClass. This can be done with the target primitive, as shown in the following example.

```
pointcut execSomeMethod1onSomeClass() :
    execution(void someMethod1()) && target(SomeClass);
```

The scope of the classes matched by a pointcut can be constrained to be the extensions of a certain class. This can be done with the within primitive. The following example shows how to constrain the scope of executions of someMethod1() to the extensions of AbstractClass.

```
pointcut execSomeMethod1onAbstractClassExtensions() :
    within(AbstractClass+) && execution(void someMethod1());
```

Advices

An *advice* is the primitive for introducing behavior at a certain pointcut. The behavior can be introduced *before*, *after*, or *around* the pointcut. Example definitions of advices are given below, using the example pointcut declarations introduced previously.

The following example shows how to introduce behavior before the pointcut execSomeMethod1() (given previously).

```
before() : execSomeMethod1() {
    /* do something prior to the method execution, e.g. */
    System.out.println("someMethod1() is going to be executed");
}
```

The following example shows how to introduce behavior after the pointcut execSomeMethod2(Type) (given previously), having access to the method parameter.

```
after(Type obj) : execSomeMethod2(Type) && args(obj) {
    /* do something with obj after the method execution, e.g. */
    System.out.println("someMethod2() was executed with " + obj);
}
```

The definition of an advice may have the pointcut definition embedded in it. The following example defines an advice in that way. Moreover, the example also shows how to gain access to the object that is returned (after execution).

```
after() returning(Type ret) : execution(Type someMethod3()) {
    /* do something with the returned object (ret), e.g. */
    System.out.println(ret);
}
```

An *around* advice implies that the introduced behavior executes instead of the behavior at the given pointcut. In around advices it is possible to use the proceed primitive for defining that the behavior at the pointcut should execute normally. The example below shows an advice that wraps the invocation of someMethod3(). It tests some condition, so that if it is true the method executes normally, otherwise it returns the object wrapped in a TypeProxy object (suppose this type to be a subtype of Type with a constructor having a parameter of type Type).

```
Type around() : execution(Type someMethod3()) {
    if(someCondition())
        return proceed(); /* does not change behavior */
    else
        return new TypeProxy(proceed());
}
```

Aspects

An *aspect* is a type of modularization unit that is used in an AOP program, in addition to classes. An aspect module is declared using the keyword aspect, and apart from a few details, it may contain everything that a class may have (i.e. attributes, methods, etc), plus pointcut definitions and advices. The following is an example aspect where all the parameters that are used in the executions of someMethod2() are stored in an array. Assume the existence of the type Type.

```
public aspect SomeAspect {
    private List<Type> paramsLog;

    public SomeAspect() {
        paramsLog = new ArrayList<Type>();
    }

    private void logParam(Type param) {
        paramsLog.add(param);
    }

    private pointcut execSomeMethod2(Type t) :
        execution(void someMethod2(Type)) && args(t);

    after(Type obj) : execSomeMethod2(Type) && args(obj) {
        logParam(obj);
    }
}
```

In AspectJ, aspects may only contain constructors that have no arguments. The reason is that they are instantiated automatically at the time of the execution of one of its advices, while the programmer cannot instantiate them manually as in the case of classes. An aspect instance is analogous to the instance of a class, i.e. an object. By default, an aspect is instantiated only once. However, it may be defined that there should occur one aspect instantiation per advice execution.

Analogously to classes, aspects may inherit from other aspects. An aspect (*subaspect*) inherits the advices, methods, and pointcuts, of its *superaspect*. Both methods and pointcuts can be overridden in subaspects, if the their visibility allows. A method in a subaspect can be overriden in the same way as in classes, and pointcut overriding is analogous.

Aspects may also be abstract, and the keyword abstract is used for that purpose, as in abstract classes. Abstract aspects may contain abstract pointcut declarations and abstract methods. The aspect shown in the previous example actuates on every execution of someMethod2(), despite the owner class. The following example presents an abstract aspect that is similar to the one given in the previous example, but the particular class that owns someMethod2() is left open to be defined by subaspects. Moreover, the advice only works for extensions of SomeAbstractClass, due to the use of within(SomeAbstractClass+).

```
public abstract aspect SomeAbstractAspect {
   /* ... */
   protected abstract pointcut targetClass();

   after(Type obj) :
      within(SomeAbstractClass+) && targetClass() &&
      execution(void someMethod2(Type)) && args(obj) {
         logParam(obj);
      }
}
```

Below we can see an example of a concrete aspect that inherits from SomeAbstractAspect. The aspect implies that only the executions of someMethod2() of SomeClass will be logged. Assume that SomeClass inherits from SomeAbstractClass.

```
public aspect SomeConcreteAspect extends SomeAbstractAspect {
   protected pointcut targetClass() : target(SomeClass);
}
```

Precedences

Different aspects may introduce behavior in the base program in the same pointcuts. In these cases, if the order of advice execution is relevant and cannot be arbitrary, aspect *precedences* may be used for determining the execution order of the advices of different aspects. Precedences can be declared using the following primitive:

```
declare precedence: AspectB, AspectA;
```

The precedence rules vary according to the advice type (i.e. before, after, around), and are not detailed here. In this example, assuming that AspectA and AspectB have *after* advices, the precedence declaration defines that AspectA has the lowest precedence, and therefore, that its advice will execute first.

3.3 Concept

This section describes the concept of specialization aspects (SAs) and application aspects (AAs) in terms of general object-oriented and aspect-oriented concepts.

Figure 3.3 presents a conceptual model that describes SAs and AAs. Recall that an SA represents a concept provided by the framework, while the AAs that inherit from that SA represent instances of that concept in a framework-based application. An SA is a module that can be either an *abstract class* or an *abstract aspect*. Although using the term “specialization aspect” for a class may sound awkward, extensions of that class are indeed intended to implement a single concern of a framework-based application. If an SA is an abstract class then it means that the concept that it represents may exist on its own, e.g. the main class of an application or a component. Otherwise, if the SA is an abstract aspect, it represents a concept that is part of another concept. With respect to framework-based applications, each concept instance is represented in one AA, which inherits from the SA that represents the concept. In case the SA is an abstract class, the AA is a *concrete class*, whereas if the SA is an abstract aspect the AA is *concrete aspect*.

An SA may have *parameters*, for which AAs provide *parameter values*. An SA may have two kinds of methods. On the one hand, there are *hidden methods*,

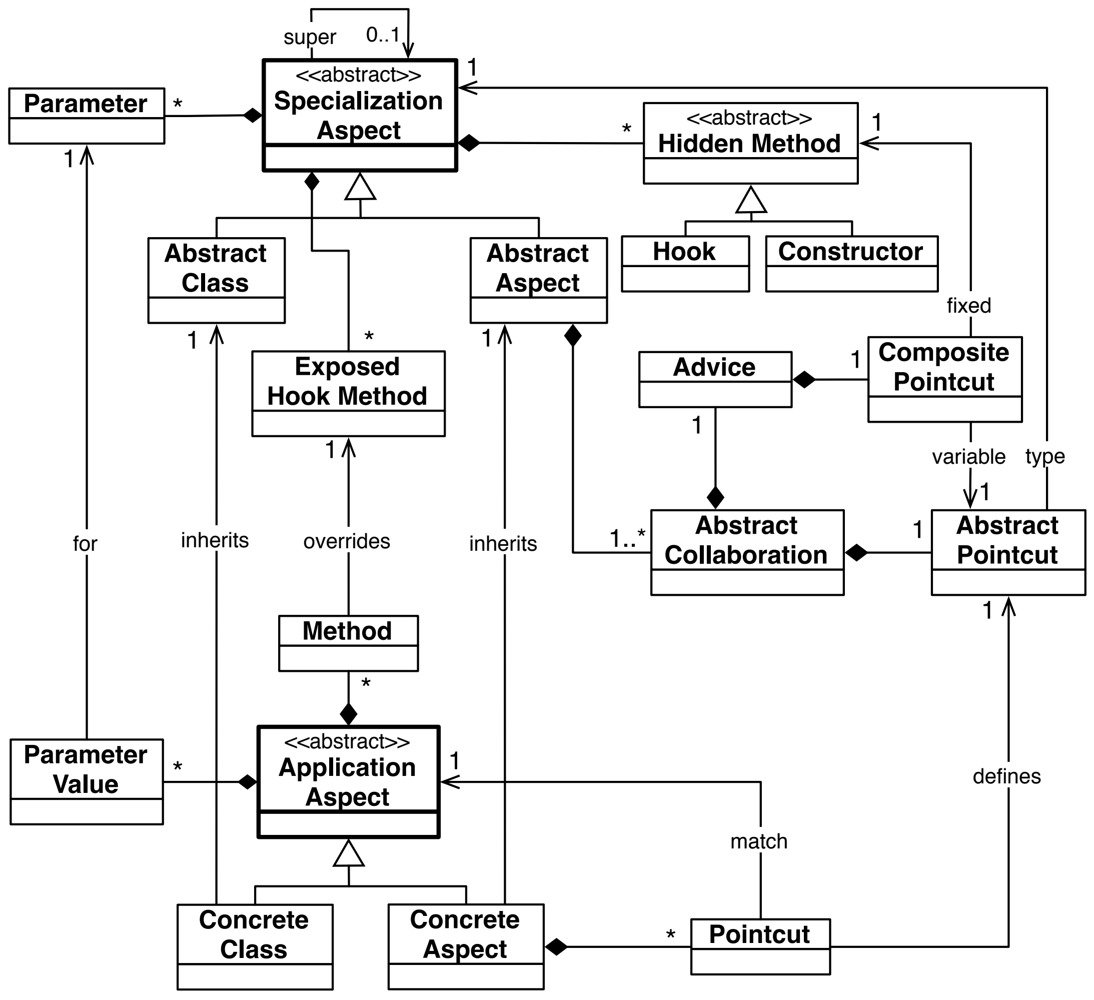


Figure 3.3: Conceptual model for specialization aspects and application aspects.

which can be either *hook methods* or *constructor methods*. Both actual constructors and methods that create objects are considered to be constructor methods. These methods are hidden from framework-based applications, while they have to be visible among the SAs. On the other hand, *exposed hook methods* are methods whose behavior is meant to be defined by framework-based applications. Hence, an AA contains *methods* that override the exposed hook methods of the SA from which it inherits.

SAs that are abstract classes can only have parameters and exposed hook methods, whereas SAs that are abstract aspects can additionally implement *ab-*

stract collaborations with other SAs. The idea is an SA to abstractly implement possible collaborations with other SAs. The collaboration participants are the AAs that inherit from the SAs. Each abstract collaboration is represented by an *advice* and an *abstract pointcut*. The advice actuates on a *composite pointcut* (Hanenberg *et al.*, 2003), which has a fixed part that intercepts a hidden method of the other SA which the SA collaborates with, and a variable part represented in the abstract pointcut. The abstract pointcut has an associated type, given as an SA, that defines that the pointcut should be made concrete by matching an AA that inherits from that SA. In the composite pointcut, the type of the abstract pointcut is always equal to the SA that owns the hidden method. The fact that the composite pointcut is based on an abstract pointcut makes the advice also abstract. The advice only takes effect if the abstract pointcut is made concrete.

An SA may inherit from another SA (super). In these cases, the sub-SAs inherit from the super SA the exposed and hidden methods, as well as the abstract collaborations. Inheritance between SAs is useful for structuring and enabling easy extensibility.

The AAs that inherit from an abstract class only have to provide the parameter values and methods, if any. On the other hand, the AAs that inherit from an abstract aspect, in addition to parameter values and methods, have to define one *pointcut* for each abstract pointcut of the SA. The pointcut has to match another AA with the correct type. The definition of pointcuts makes the abstract collaborations concrete, given that the variable part becomes bound.

3.4 Capturing Framework Hot Spots

This section explains how framework hot spots can be expressed in SAs. It is demonstrated how the basic adaptation mechanisms are captured with SAs, and how such SAs can be implemented in AspectJ.

The different adaptation mechanisms described in Section 2.1 imply that different framework concepts are instantiated in different ways. For instance, a certain framework concept may be instantiated by extending a class and overriding a hook method, while another framework concept may be instantiated by composing a parameterized object with another object exposed by the framework.

In contrast to the conventional adaptation mechanisms, SAs provide a uniform way of instantiating the concepts provided by the framework.

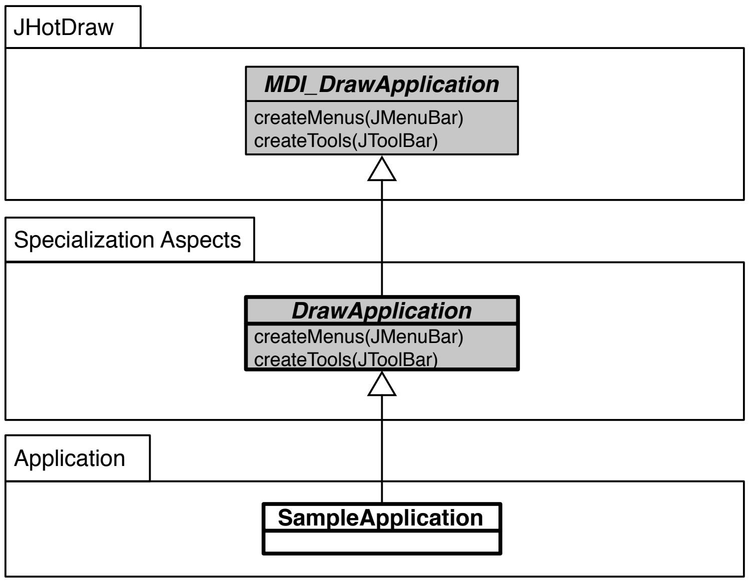
The following sections address the several framework adaptation mechanisms, using hot spots of the JHotDraw framework as examples. Throughout the several figures that illustrate the solutions, framework modules are depicted in gray, application modules are depicted in white, and stereotyped dependencies between AAs denote pointcut definitions. The code examples are illustrated with figures that contain UML class diagrams representing the different layers in the packages: JHotDraw framework, specialization aspects, and application aspects. When there is need to disambiguate, the prefix sas is used to refer to a member of the specialization aspects package, while the prefix jhd is used to refer to a member of the JHotDraw framework package.

3.4.1 Class inheritance and object parameterization

Inheritance is a common adaptation mechanism in frameworks, and solutions that apply the Template Method pattern (Gamma *et al.*, 1995) occur very frequently. These cases require application classes to inherit from a framework class, which is typically abstract, and to override hook methods.

In JHotDraw, the drawing application is represented by the abstract class MDI_DrawApplication, which framework-based applications may subclass and override several hook methods. For the purpose of illustrating specialization aspects, consider from now on that those hook methods are createMenus(JMenuBar) and createTools(JToolBar), for adapting the application menus and creation tools, respectively.

In order to address the concept of *draw application* one may provide an SA as shown in Figure 3.4. The SA is an abstract class that subclasses the framework class MDI_DrawApplication. However, notice that the two hook methods are overridden (preserving the same behavior as in MDI_DrawApplication) and declared as final, preventing the subclasses of DrawApplication of overriding them. Given that the given hook methods pertain to different concepts, *menu* and *creation tool*, they should be handled by different SAs, which are going to complete the behavior of the hook methods.



```
public abstract class DrawApplication extends MDI_DrawApplication {
 public DrawApplication(String name) {
  super(name);
 }

 protected final void createMenus(JMenuBar mb) {
  super.createMenus(mb);
 }

 protected final void createTools(JToolBar palette) {
  super.createTools(palette);
 }
}
```

```
public class SampleApplication extends DrawApplication {
 public SampleApplication() {
  super("Sample");
 }
}
```

Figure 3.4: Specialization aspect capturing class inheritance.

Figure 3.4 also presents the class SampleApplication, which is an AA that only defines its constructor (as required by the SA). Notice that the class does not define the hook methods. Given that such hook methods cannot be overridden, they are no longer in control of the framework-based application. The next subsections present SAs that that handle the behavior of such hook methods.

The application objects that are handled by AAs may need to be parameterized. This can be achieved by having the SAs with a constructor, whose parameter values are used for instantiating the objects. In turn, the AAs that inherit from the SA become forced to define a constructor. The body of such a constructor should be a call to the super constructor with the desired parameter values, as in the given example. More examples of parameterization are presented in the following sections.

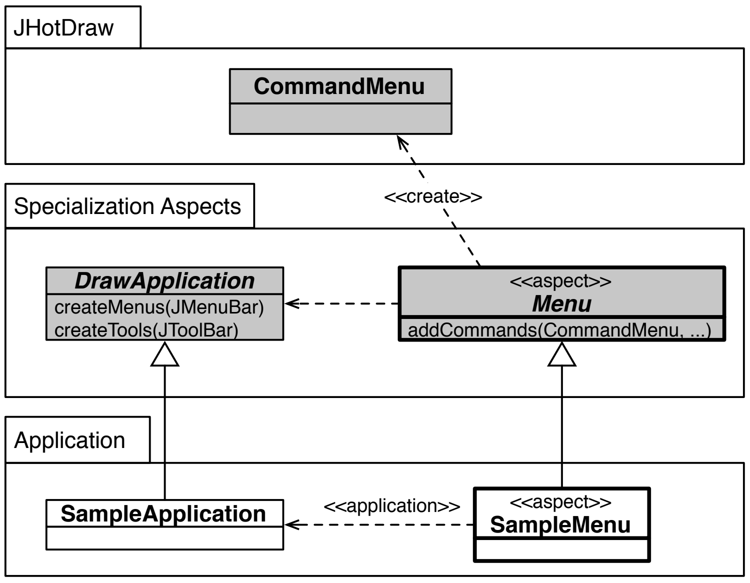
In terms of the concepts of Figure 3.3, the example in this subsection presented an SA (DrawApplication) that is an abstract class with one parameter (name), defining two hidden hook methods (createMenus(..) and createTools(..)). With respect to the framework-based application, the example presented an AA (SampleApplication) that is a concrete class that inherits from the SA, providing the required parameter value ("Sample").

3.4.2 Hook method overriding

When having adaptation based on inheritance, hook methods are an essential means to support variation in subclasses. The previous subsection introduced the SA DrawApplication, which defines a hook method for handling the menus. In order to explain how SAs can capture hook method overriding, this subsection presents an SA for handling the behavior of that hook method.

In the JHotDraw framework, a menu is represented by an instance of CommandMenu. Menus can be plugged into a framework-based application by composing CommandMenu objects in the JMenuBar object exposed by the hook method createMenus(..), introduced in the previous section.

In order to develop an SA for addressing the concept of *menu* one has to implement an abstract collaboration that plugs CommandMenu objects, within the body of the createMenus(..) of a framework-based application. Such an SA



```
public abstract aspect Menu {
 private String name;

 public Menu(String name) {
  this.name = name;
 }

 protected abstract pointcut application();

 after(DrawApplication application, JMenuBar mb) :
  within(DrawApplication+) && application() &&
  execution(void DrawApplication.createMenus(JMenuBar)) &&
  this(application) && args(mb) {
   CommandMenu menu = new CommandMenu(name);
   mb.add(menu);
   addCommands(menu, application);
  }

 void addCommands(CommandMenu menu, DrawApplication appplication) {
 }
}
```

```
public aspect SampleMenu extends Menu {
 public SampleMenu() {
  super("SampleMenu");
 }

 protected pointcut application() : target(SampleApplication);
}
```

Figure 3.5: Specialization aspect capturing hook method overriding.

has to be an abstract aspect, given that it is necessary to introduce behavior at another AA.

Figure 3.5 presents an SA Menu for plugging menus into a JHotDraw-based application, assuming the existence of the SA DrawApplication. The abstract aspect defines a constructor so that a menu name is passed when instantiating the aspect. Moreover, it defines an abstract collaboration with DrawApplication. Such an abstract collaboration is composed of an abstract pointcut application(), which is to be defined in the AAs that inherit from the SA, and an advice on a composite pointcut. The composite pointcut intersects the pointcut on the method DrawApplication.createMenus(JMenuBar) with application(). The figure shows an example AA, SampleMenu, which defines the pointcut application() on SampleApplication. Notice that in this solution CommandMenu objects can be cohesively plugged in the application via an AA, and no modification or inspection of SampleApplication is required.

Due to the primitive within(DrawApplication+) the advice may only take effect in the subclasses of DrawApplication. The primitives this(application) and args(mb) together with the declaration after(DrawApplication application, JMenuBar mb) enable the aspect to gain access to the instance of DrawApplication where the method is invoked and the JMenuBar object that is passed to the hook method. The body of the advice creates a CommandMenu object using the name attribute, plugs it into the JMenuBar object, and finally, it invokes addCommands(..), which is a hook method for plugging commands into the menu. An SA that implements an abstract collaboration involving this hook method is presented later on.

The hook method createMenus(...), which in a conventional reuse interface would be defined by applications, is no longer visible to application developers. Finally, the type JMenuBar is no longer relevant to framework-based applications, and therefore, it is excluded from the new reuse interface based on SAs.

In terms of the concepts of Figure 3.3, the example in this subsection presented an SA (Menu), which is an abstract aspect with one parameter (name). Such an abstract aspect defines an abstract collaboration composed of one abstract pointcut (application()) and advice defined on a composite pointcut that intersects the abstract pointcut with the execution of the hidden hook method of another SA

(DrawApplication.createMenus(..)). With respect to the framework-based application, the example presented an AA (SampleMenu), which is a concrete aspect that inherits from the SA that provides the required parameter value ("SampleMenu") and a contrete pointcut (application()) that defines the abstract pointcut of the SA.

3.4.3 Structuring specialization aspects with inheritance

In order to avoid code duplication and to promote extensibility, inheritance may be used to structure SAs. On the one hand, frameworks may offer an hierarchies of types whose objects may be plugged into framework-based applications. On the other hand, SAs may be structured using inheritance due to certain related objects having a similar way of being plugged into a framework-based application. In both cases, using inheritance relationships between SAs can be beneficial.

3.4.3.1 Hierarchies of pluggable objects

In the JHotDraw framework there are two framework types, Figure and ConnectionFigure, for representing *nodes* and *connections*, respectively. The framework also provides several implementations of these types, which are organized in class hierarchies.

When using SAs, a suitable solution is to have a hierarchy of SAs, where the top-most SA implements the abstract collaboration against the framework types, while several subaspects of it (i.e. other SAs) provide the different implementations of the framework types. Figure 3.6 illustrates this solution. Figure is the top-most SA that implements an abstract collaboration with DrawApplication, which implies the creation of an instance of the figure when an instance of DrawApplication is created. The AAs extend the SA that implements the desired figure and define the pointcut on the application where the figure is going to be included. Figure 3.7 presents the code corresponding to the solution depicted in the diagram.

In terms of the concepts of Figure 3.3, the given example presents several SAs that have another SA as its super aspect. However, only the top-most SA

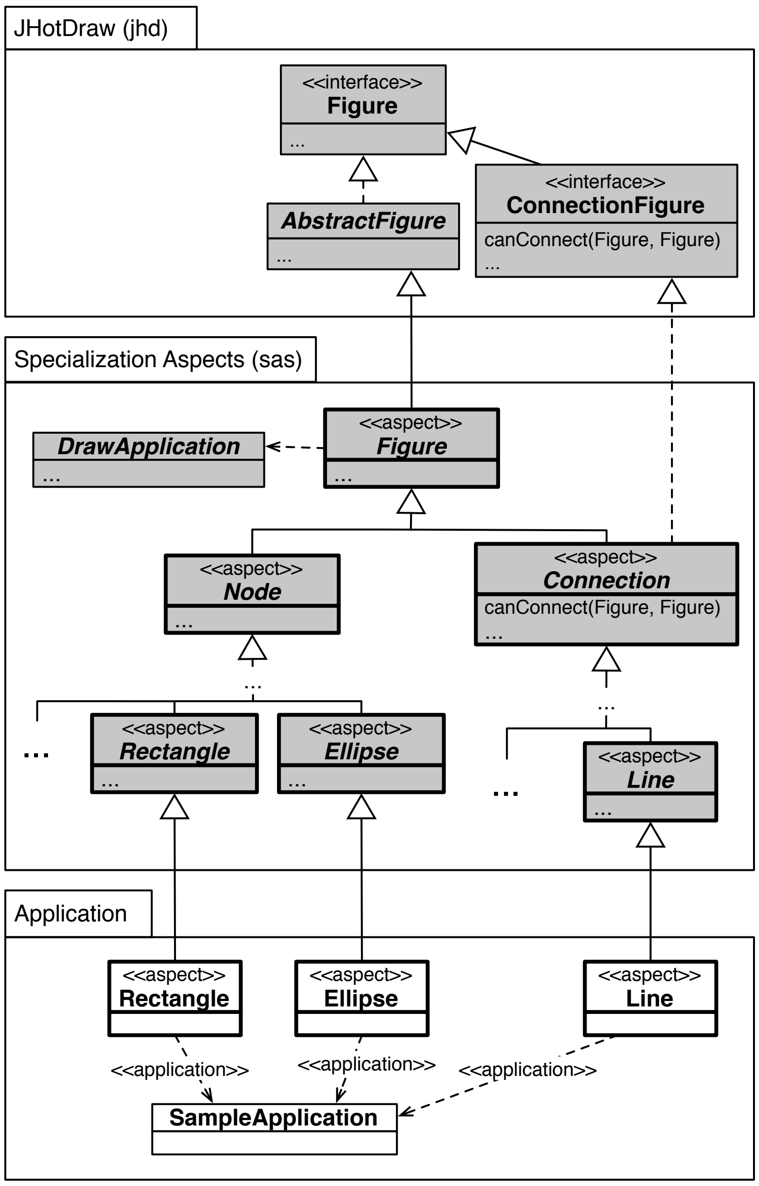


Figure 3.6: Hierarchy of pluggable objects (diagram).

```
public abstract aspect sas.Figure extends AbstractFigure {
 protected abstract pointcut application();

 after() :
  within(DrawApplication+) && execution(*.new()) && application() { }
}
```

```
public abstract aspect Node extends sas.Figure {
 // Implementation of jhd.Figure interface methods
}
```

```
public abstract aspect Connection
extends sas.Figure implements ConnectionFigure {
 public final boolean canConnect(jhd.Figure start, jhd.Figure end) {
  return false;
 }
}
```

```
public abstract aspect AttributeFigure extends Node {/* ... */}
```

```
public abstract aspect Rectangle extends AttributeFigure {
 // implementation of rectangle specifics
}
```

```
public abstract aspect Ellipse extends AttributeFigure {
 // implementation of ellipse specifics
}
```

```
public abstract aspect PolyLineFigure extends Connection {/* ... */}
```

```
public abstract aspect Line extends PolyLineFigure {
 // implementation of line specifics
}
```

```
public aspect Rectangle extends sas.Rectangle {
 public pointcut application() : target(SampleApplication);
}
```

```
public aspect Ellipse extends sas.Ellipse {
 public pointcut application() : target(SampleApplication);
}
```

```
public aspect Line extends sas.Line {
 protected pointcut application() : target(SampleApplication);
}
```

Figure 3.7: Hierarchy of pluggable objects (code).

implements an abstract collaboration (with DrawApplication) and defines an abstract pointcut, while all the other descendant SAs in the hierarchy inherit such abstract collaboration and abstract pointcut. With respect to the framework-based application, the example presents three AAs that inherit from leafs of the hierarchy of SAs. Although not exemplified here, the application could define its own figures and connection figures by having AAs that inherit directly from the SAs Node and Connection, respectively.

3.4.3.2 Generalizing common behavior

In the JHotDraw framework, the way to include a *node tools* and *connection tools* is very similar. In both cases adaptation should take place within the hook method createTools(..) that was presented earlier. The difference is that they require the creation of objects of different types. For a node tool a CreationTool object must be created, while for a connection tool a ConnectionTool object must be created. Moreover, the *undo* support is achieved in the same way for both cases. If we would develop two SAs, one for each case, both implementing an abstract collaboration with DrawApplication for completing the hook method createTools(..), we would end up having SAs with very similar advices.

If there are SAs whose advices are similar, the commonality may be factored out to an SA from which the related SAs inherit from. In the given case, since the difference resides in the object that is created, one may develop an SA with a hidden hook method responsible for creating such object. Then we can have two other SAs that inherit from such SA, overriding the hook method according to the object that has to be created. Figure 3.8 illustrates this solution.

In the given solution we can see the SA CreationTool depending on the SA DrawApplication, given that it implements an abstract collaboration involving createTools(..), and depending on the framework type Tool and on the framework class UndoableTool. This SA has two SAs inheriting from it for addressing the two kinds of tool. The SA NodeTool depends on the SA Node (given previously) and on the framework class CreationTool, while the SA ConnectionTool depends on the SA Connection (given previously) and on the framework class ConnectionTool.

The code that implements the solution depicted in Figure 3.8 is split into Figures 3.9 and 3.10.

In terms of the concepts of Figure 3.3, the given example presents two related SAs (NodeTool and ConnectionTool) that inherit from the same SA (CreationTool). While both SAs inherit the abstract collaboration and abstract pointcut defined in the super aspect, each one also defines its own abstract collaboration and abstract pointcut. With respect to the framework-based application, the example presents three AAs (RectangleTool, EllipseTool, and LineTool) that inherit from leafs of the hierarchy of SAs.

3.4.4 Object composition and interface implementation

In addition to class inheritance, framework adaptation may rely extensively on compositions of objects that are instantiated within the code of a framework-based application. For instance, in the JHotDraw framework one may plug *commands* into a *menu* by composing Command objects using the method CommandMenu.add(Command). Moreover, frameworks may allow framework-based applications to provide their own components, which have to conform to framework interfaces. Such components are then plugged in the application through hook methods and/or object composition.

An SA for supporting the inclusion of menus was given previously. Such an SA was the abstract aspect Menu, and recall that it defined a hook method addCommands(CommandMenu, DrawApplication) for enabling commands to be added to the menu. The JHotDraw framework has the interface Command for representing commands and provides an abstract implementation of a command in the class AbstractCommand. Such an abstract implementation enables the development of new commands just by giving the command behavior in the method execute().

This section presents an SA for supporting the inclusion of commands, assuming the existence of the SA Menu. The solution is based on having an aspect that implements an abstract collaboration with the SA Menu for completing the body of the hook method addCommands(..). Moreover, this section also shows an example where AAs provide method implementations (the command behavior).

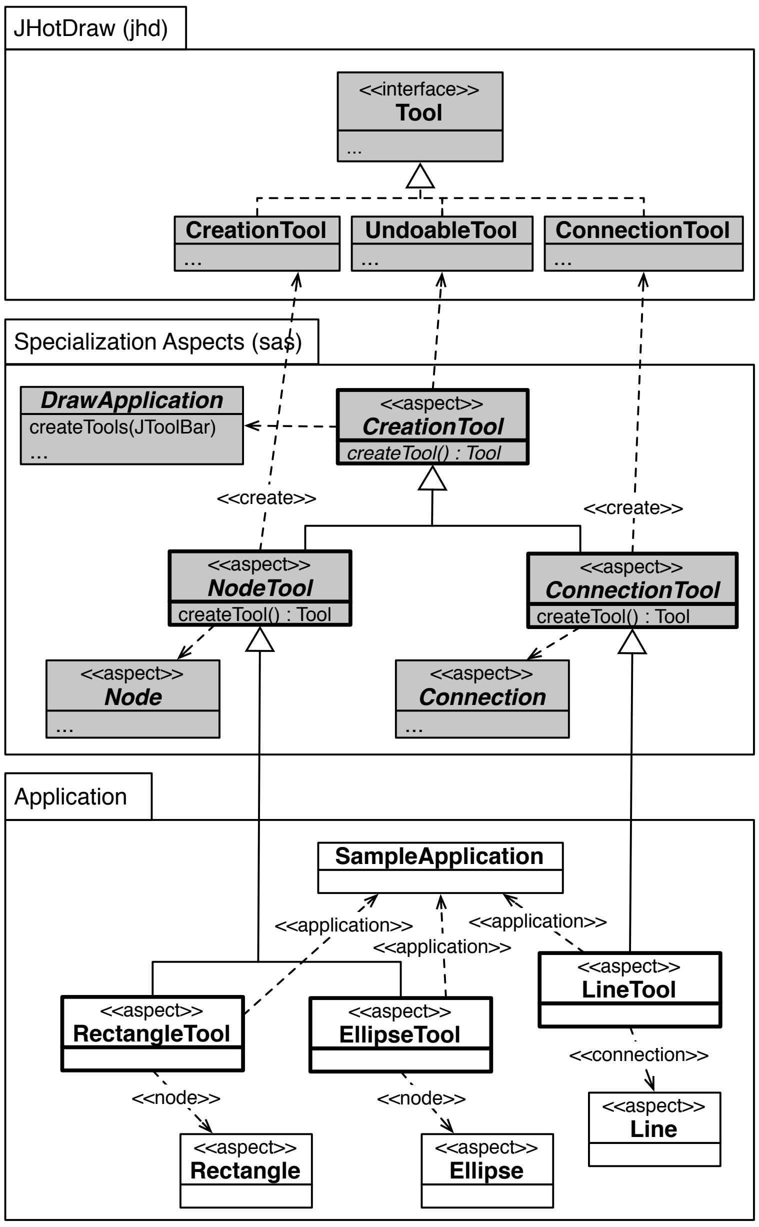


Figure 3.8: Structuring specialization aspects with inheritance (diagram).

```
public abstract aspect CreationTool {
 private String iconPath;
 private String toolTip;
 private boolean undo;

 public CreationTool(String iconPath, String toolTip, boolean undo) {
  this.iconPath = iconPath;
  this.toolTip = toolTip;
  this.undo = undo;
 }

 protected abstract pointcut application();

 after(DrawApplication app, JToolBar toolbar) :
  within(DrawApplication+) && application() &&
  execution(void DrawApplication.createTools(JToolBar)) &&
  this(app) && args(toolbar) {
   Tool tool = createTool(app);
   if(undo)
    tool = new UndoableTool(tool);
   toolbar.add(app.createToolButton(iconPath, toolTip, tool));
  }

 protected abstract Tool createTool(DrawApplication application);
}
```

```
public abstract aspect NodeTool extends sas.CreationTool {
 private Figure figure;

 public NodeTool(String iconPath, String toolTip, boolean undo) {
  super(iconPath, toolTip, undo);
 }

 protected abstract pointcut node();

 after(Node node) :
  execution(Node.new()) && node() && this(node) {
   figure = node;
  }

 protected Tool createTool(DrawApplication application) {
  return new jhd.CreationTool(application, figure);
 }
}
```

Figure 3.9: Structuring specialization aspects with inheritance (code, part 1).

```
public abstract aspect ConnectionTool extends sas.CreationTool {
 private ConnectionFigure figure;

 public ConnectionTool(String icon, String toolTip, boolean undo) {
  super(icon, toolTip, undo);
 }

 protected abstract pointcut connection();

 after(Connection connection) :
  execution(Connection.new()) && connection() && this(connection) {
   figure = connection;
  }

 protected Tool createTool(DrawApplication application) {
  return new jhd.ConnectionTool(application, figure);
 }
}
```

```
public aspect RectangleTool extends NodeTool {
 public RectangleTool() {
  super("RECT", "Default Rectangle", false);
 }

 protected pointcut application() : target(SampleApplication);
 protected pointcut node() : target(Rectangle);
}
```

```
public aspect EllipseTool extends NodeTool {
 public EllipseTool() {
  super("ELLIPSE", "Default Ellipse", true);
 }

 protected pointcut application() : target(SampleApplication);
 protected pointcut node() : target(Ellipse);
}
```

```
public aspect LineTool extends sas.ConnectionTool {
 public LineTool() {
  super("LINE","Default Line Tool", false);
 }

 protected pointcut application() : target(SampleApplication);
 protected pointcut connection() : target(Line);
}
```

Figure 3.10: Structuring specialization aspects with inheritance (code, part 2).

The proposed solution is illustrated in Figure 3.11, where we have an SA Command that depends on the SA Menu, and as an example of an application-specific command, the AA BlueBackgroundCommand that defines the abstract method action(..) of the SA.

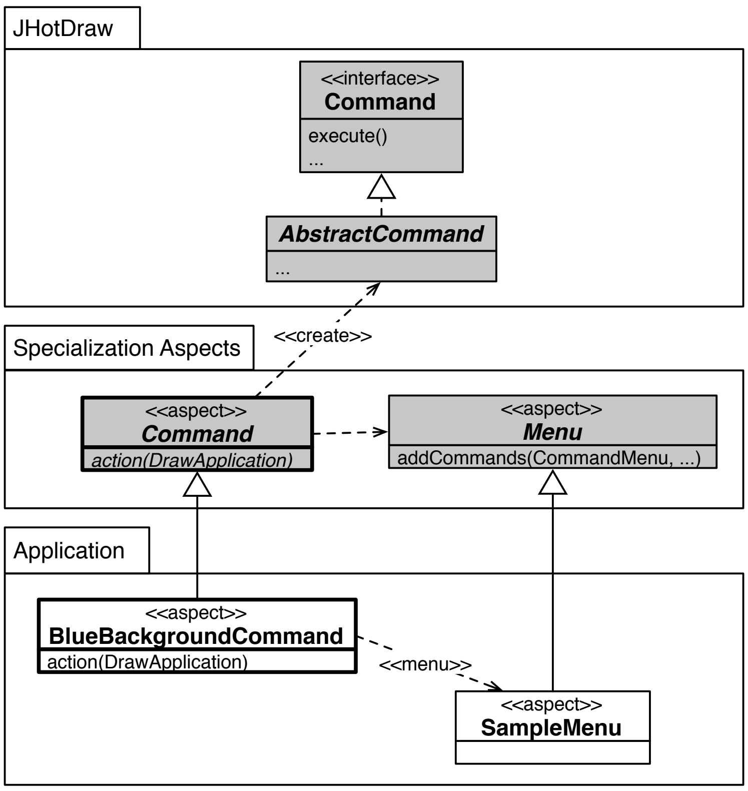


Figure 3.11: Specialization aspect capturing object composition (diagram).

The implementation of the proposed solution is given in Figure 3.12. The advice plugs an AbstractCommand object into the CommandMenu object captured in the execution of addCommands(CommandMenu, DrawApplication) of the AA to be defined by the pointcut menu(). Notice in the SA the existence of the abstract method action(DrawApplication), which is used to create the AbstractCommand object. It is the responsibility of the AAs to define that method. As in previous examples, the SA defines a parameter, which in this case is the name of the command. With respect to the framework-based application, the implementation of the AA BlueBackgroundCommand provides the command behavior in the method action(..) and defines the pointcut menu() on the AA SampleMenu (introduced earlier).

In terms of the concepts of Figure 3.3, the given example presents an SA that is an abstract aspect, implementing an abstract collaboration with the SA Menu. Such collaboration relies on the composite pointcut formed by the intersection of the pointcut menu() with the execution of the hidden hook method Menu.addCommands(..). Moreover, the SA defines the abstract method action(..), which is an exposed hook method. With respect to the framework-based application, in addition to the constructor and poincut definition, the AA BlueBackgroundCommand defines the exposed hook method.

3.4.5 Implementing relationships with specialization aspects

A framework has its domain-specific concepts and relationships. For instance, in the JHotDraw framework there are the concepts of node figure and connection figure, where each connection figure may connect valid source-target pairs of node figures. Therefore, for each valid connection that one wants to define in an application, there is a relationship between a connection figure, a source node figure, and a target node figure. JHotDraw enables the definition of valid connections by means of overriding the method ConnectionFigure.canConnect(Figure, Figure). At runtime, when the user is attempting to connect a source and a target figure, this method is executed to determine if the figures can be connected.

The design solution adopted in JHotDraw forces the definition of valid connection relationships to be tangled with the connection figures' classes. This subsection shows how such relationships can be modularized using specialization aspects. The solution is based on having an SA that defines three abstract collaborations. One collaboration is with the SA ConnectionFigure, for defining for which connection figure is associated to a valid connection. The other two collaborations are with the SA Node, for defining the source and the target node figure that can be connected. The proposed solution is illustrated in Figure 3.13, where we have an SA ValidConnection depending on the SAs Node and Connection (given previously). Framework-based applications define each valid connection in its own AA, as in the example given in the figure, where an AA RectangleToEllipse

```
public abstract aspect Command {
 private String name;

 public Command(String name) {
  this.name = name;
 }

 protected abstract pointcut menu();

 public abstract void action(DrawApplication application);

 after(Menu menu, CommandMenu commandMenu, DrawApplication app) :
  within(Menu+) && menu() &&
  execution(void addCommands(CommandMenu, DrawApplication)) &&
  this(menu) && args(commandMenu, app) {
   final DrawApplication application = app;
   AbstractCommand cmd = new AbstractCommand(name, app) {
    public void execute() {
     action(application);
    }
   };
  commandMenu.add(cmd);
 }
}
```

```
public aspect BlueBackgroundCommand extends sas.Command {
 public BlueBackgroundCommand() {
  super("Color background in blue");
 }

 protected pointcut menu() : target(SampleMenu);

 public void action(DrawApplication application) {
  application.view().setBackground(Color.BLUE);
 }
}
```

Figure 3.12: Specialization aspect capturing object composition (code).

defines a valid connection from the rectangle to the ellipse figures, for the line connection figure.

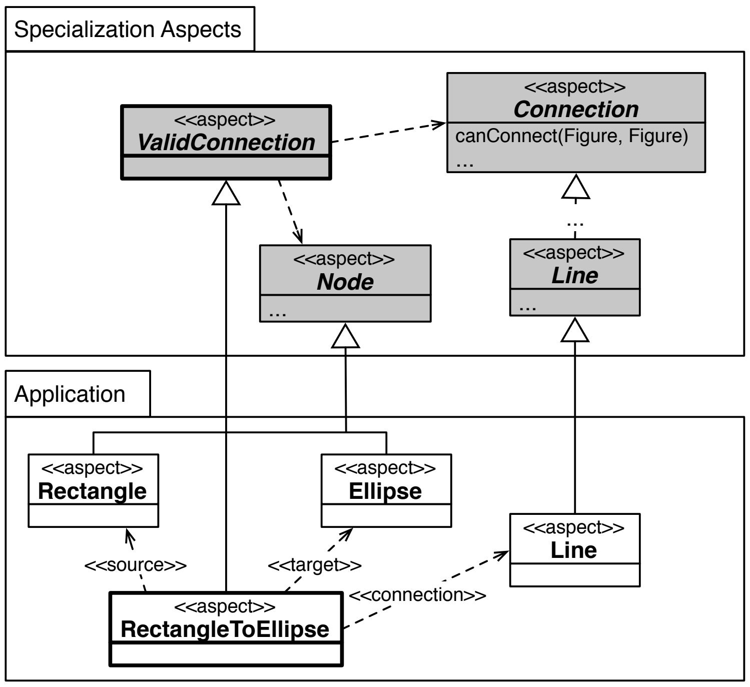


Figure 3.13: Specialization aspect capturing a relationship (diagram).

The implementation of the proposed solution is given in Figure 3.14. The abstract aspect ValidConnection defines three abstract collaborations, and therefore, three abstract pointcuts and three advices. The two first advices are very similar. They capture the instantiation of AAs that inherit from Node, so that the figure class of the source and target nodes is stored in the attributes sourceClass and targetClass, respectively. The third advice completes the body of the hidden hook method Connection.canConnect(Figure, Figure). The classes of the two figure objects are checked against the classes stored in the attributes, and if they match the method returns true. Otherwise, the method execution proceeds, since they might exist other valid connections that have to be checked. With respect to the framework-based application, the figure presents an AA RectangleToEllipse that implements the valid connection by defining the pointcuts on the AAs that are associated with the figures.

```
public abstract aspect ValidConnection {
 private Class sourceClass;
 private Class targetClass;

 protected abstract pointcut connection();
 protected abstract pointcut source();
 protected abstract pointcut target_();

 after(Node figure) :
  execution(Node.new()) && source() && this(figure) {
   sourceClass = figure.getClass();
  }

 after(Node figure) :
  execution(Node.new()) && target_() && this(figure) {
   targetClass = figure.getClass();
  }

 boolean around(jhd.Figure sourceFig, jhd.Figure targetFig) :
  within(ConnectionFigure+) && connection() &&
  execution(boolean Connection.canConnect(jhd.Figure, jhd.Figure)) &&
  args(sourceFig,targetFig) {

  if(sourceClass.equals(sourceFig.getClass()) &&
     targetClass.equals((targetFig.getClass())))
   return true;
  else
   return proceed(sourceFig,targetFig);
 }
}
```

```
public aspect RectangleToEllipse extends ValidConnection {
 protected pointcut connection() : target(Line);
 protected pointcut source() : target(Rectangle);
 protected pointcut target_() : target(Ellipse);
}
```

Figure 3.14: Specialization aspect capturing a relationship (code).

The given mechanism enables the encapsulation of relationships without knowing where and how the involved objects are instantiated. When using conventional adaptation this kind of encapsulation would not be possible. Notice that such compositions are an increment that does not modify or require the inspection of other AAs.

In terms of the concepts of Figure 3.3, the given example presents an SA that is an abstract aspect, implementing abstract collaborations with the SAs Node and Connection. The collaboration with Connection is implemented in a similar way to other examples introduced earlier. The other two collaborations differ in the fact that they rely on capturing the execution of a hidden constructor method (the constructor of Node).

3.4.6 Ordering of application aspects

When having several AAs that inherit from the same SA, and whose pointcuts definitions are matching the same modules, the advices are interfering in common pointcuts. In these cases, it might be necessary for an application to define aspect precedences to determine in which order the advices are applied. For instance, the order of the creation tools in the toolbar of a JHotDraw-based application is determined by the order by which the creation tools are composed within the associated hook method. When using the SAs, one could define the following order (1) rectangle tool, (2) ellipse tool, and (3) line tool, by defining the aspect given in Figure 3.15. The AA RectangleTool has the highest precedence and the AA LineTool has the lowest. The aspects with higher precedence are weaved first in the case of after advices, and therefore, we would achieve the intended order.

```
public aspect ToolOrder {
 declare precedence:  LineTool, EllipseTool, RectangleTool;
}
```

Figure 3.15: Aspect for defining the order of application aspects.

3.5 Benefits

This section summarizes the benefits of expressing framework hot spots by means of SAs. Such benefits are related with having reuse interfaces that can be used at a higher abstraction level than conventional ones. The following advantages, illustrated by the examples of Section 3.4, justify this claim.

- *Modular and uniform reuse interface.* Framework hot spots are represented explicitly and uniformly in SAs, having a one-to-one mapping between SAs an concepts provided by the framework.
- *Modular framework-based applications.* Framework-based applications can be developed in an incremental way. Each AA is an increment that does not require modifications or inspection of other AAs. Object compositions and relationships can also be encapsulated as AAs.
- *Less hook methods.* Framework-based applications can be developed without dealing with as many hook methods as in conventional solutions. This contributes to have a *narrow inheritance interface*, a principle that states that only a few hook methods should be required to be given per each application class (Weinand *et al.*, 1989).
- *Less application-relevant methods.* Framework-based applications can be built using less framework methods than in conventional solutions. This may imply that whole framework classes/interfaces will become irrelevant to applications.

3.6 Discussion

Frameworks tend to evolve from white-box to black-box (Roberts & Johnson, 1997). By developing a reuse interface based on SAs, the framework evolves in this direction, too. However, SAs enable more high-level framework usage than the reuse interface of a conventional black-box framework, given the benefits explained in Section 3.5.

The proposed concept of SA only requires a relatively small subset of the AOP primitives that a language such as AspectJ offers. These are abstract aspects, abstract pointcuts, and method join points. Most of the primitives offered by AspectJ are not necessary.

AspectJ was found suitable for developing SAs for Java frameworks. However, its primitives are not particularly "tuned" for this purpose, and thus the advice definitions may seem inelegant. Although dedicated language primitives could be developed for the specific purpose of SAs, they were found non-essential given that they would only enable to have more elegant code.

Besides technical issues regarding the chosen AOP language, SAs may be hard to develop due to their unconventional design nature. Chapter 4 presents a pattern language composed of several design patterns that aid on the development of SAs. A concrete running example is also used throughout the pattern descriptions in order to illustrate how a conventional reuse interface can be enhanced with SAs.

Chapter 4

Patterns for Framework Specialization Aspects

The concept of framework specialization aspects was addressed in Chapter 3. This chapter presents design patterns for assisting domain engineers in the development of framework specialization aspects. The design patterns are presented as a *pattern language* (Alexander *et al.*, 1977) referred to as MODULAR HOT SPOTS. The patterns of the language can be used together to develop specialization aspects for an existing framework.

Section 4.1 introduces a toy framework that is used as a running example for illustrating the patterns. Section 4.2 presents an overview of the pattern language. Section 4.3 presents an AspectJ idiom that is used in the implementation of the patterns. Sections 4.4-4.9 present the patterns that constitute the pattern language. Section 4.10 revisits the example framework taking into account the new reuse interface that resulted from all the examples given throughout the patterns. Finally, Section 4.11 discusses the usefulness of the patterns for building F-DSLs.

4.1 Example Framework

This section introduces a simple example of a framework, which can be used to build GUI *applications*. A GUI application has *actions* that can be triggered by the UI elements. The action can be either application-specific or provided by the framework. An application may have *menus*, which may contain submenus. The menus may contain either items that trigger application actions or other menus (i.e. the submenus). Implementation-wise there is no distinction between a menu and a submenu (i.e. they are represented by the same class). Figure 4.1 presents an UML class diagram describing the framework's reuse interface, showing the elements that an application developer needs to deal with in order to build an application. Below we present Java code that implements an example framework-based application using the conventional reuse interface.

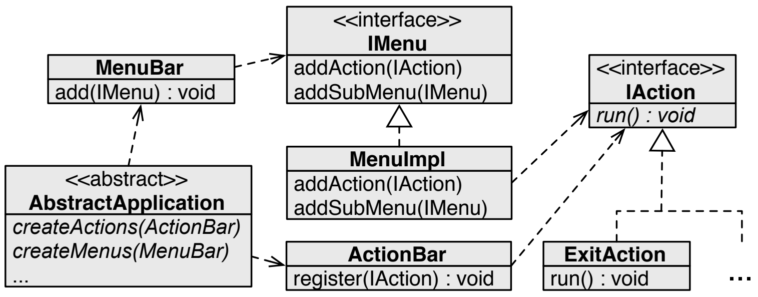


Figure 4.1: Reuse interface of the example framework.

```
public class ExampleApplication extends AbstractApplication {
  private IAction myaction, exitaction;

  protected void createActions(ActionBar a_bar) {
    myaction = new ExampleAction();
    a_bar.register(myaction);
    exitaction = new ExitAction();
    a_bar.register(exitaction);
  }

  protected void createMenus(MenuBar m_bar) {
    IMenu menu1 = new MenuImpl("Menu1");
    menu1.addAction(exitaction);
    IMenu menu2 = new MenuImpl("Menu2");
    menu2.addAction(myaction);
    menu1.addSubMenu(menu2);
    m_bar.add(menu1);
  }
}
```

```
public class ExampleAction implements IAction {
  void run() {
    // do something
  }
}
```

The main class is a subclass of AbstractApplication. Application developers must be aware that createActions() is executed before createMenus(). The sample framework-based application has two actions, an application-specific one, ExampleAction, and the framework-provided ExitAction. It has a "Menu1", which has the exit action and a submenu "Menu2" that has the application-specific action.

Usage scenarios

The following list presents a set of scenarios where application developers may be faced with difficulties. Each of the scenarios is associated with a goal.

- *Scenario 1, plugging menus.* The application concept *menu* is represented directly by the interface IMenu, which MenuImpl implements. Therefore, it should be easy for an application developer to locate it. However, once the interface/class is known, it is necessary to find out how to plug the menu into the application. Given that the *application* concept is represented abstractly by the AbstractApplication class, one would go to inspect that class, and then realize that there is a hook method for the intended purpose (i.e. createMenus(...)). Plugging in the menu involves modifying the method body, which may have existing statements. Therefore, in order to implement the goal of plugging a menu, one has to "interfere" with statements pertaining to other goals (i.e. other menus and their contents).

- *Scenario 2, menu context.* The application concept *menu* can be used in two different contexts, either as an *application* menu or as *submenu* of another menu. As explained in Scenario 1, by knowing IMenu and MenuImpl one does not know where and how the menus can be plugged. If one has an existing application menu *m1* and wants that menu to become a submenu of another menu *m2*, besides needing to understand the subclass of AbstractApplication (Scenario 1) to remove the statement that plugs in the menu, there is need to locate where *m2* is instantiated and to know its interface to add *m1* to it. Therefore, changing the context of an existing menu requires changes both in statements pertaining to the original context and in statements pertaining to the new context.

- *Scenario 3, associating actions.* The application concept *action* is represented by the interface IAction. Actions may be associated to *menus*. Suppose that there is an existing application with an action *a* and a menu *m*, and that one wants to associate *a* with *m*. In order to do so, one has to inspect the hook method createActions() of the subclass of AbstractApplication to find out the instance of *a*, and then, to modify the hook method createMenus() by finding the instance of *m* and adding a statement that associates *a* to it. Therefore, an association between two application elements involves two parts of a module (the subclass of AbstractApplication) which is not directly related to those elements.

Each of the given scenarios can be improved by applying the MODULAR HOT SPOTS pattern language. When addressing a scenario by applying a pattern, it might happen that a new problem arises. In such cases, there are other patterns for overcoming these problems.

4.2 Pattern Language Overview

Figure 4.2 gives an overview of MODULAR HOT SPOTS. The diagram contains related patterns and idioms represented in white, while the actual patterns/idioms of the language are represented in gray. Design patterns are represented as ellipses, whereas AspectJ idioms are represented as circles.

Hot spots based on TEMPLATE METHOD (Gamma *et al.*, 1995) are typical starting points for applying the pattern language. It is common that an application framework applies at least one TEMPLATE METHOD on the main class that initializes the application. A TEMPLATE METHOD has one or more hook

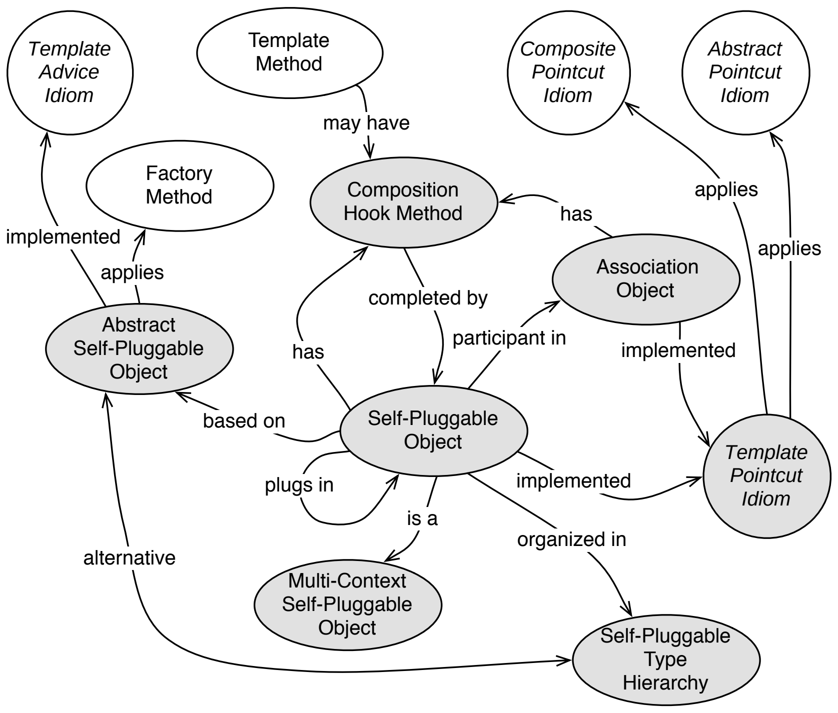


Figure 4.2: Modular Hot Spots pattern language.

methods, which have to be overridden by application developers. A Composition Hook Method (Section 4.4) is a hook method that exposes an object instantiated by the framework as a parameter, with the purpose of enabling applications to plug objects in the exposed object. While this pattern is not related with the development of Modular Hot Spots directly, it describes a common solution that hints where it is suitable to have a Self-Pluggable Object (Section 4.5). As we will see, Composition Hook Methods are "predictable" and can be completed by a Self-Pluggable Object, after which the application developer no longer has to deal with those hook methods.

A Self-Pluggable Object is a hot spot that enables its adaptations to localize both the creation of an object representing an application element and its composition with another application element. It may be plugged into another Self-Pluggable Object, it may have Composition Hook Methods itself, and it can be implemented using a Template Pointcut (Section 4.3). A Template Pointcut is an AspectJ idiom that combines the idioms Abstract Pointcut and Composite Pointcut (Hanenberg *et al.*, 2003). The Self-Pluggable Object pattern is suitable for improving the *plugging menus* scenario described in Section 4.1.

A Multi-Context Self-Pluggable Object (Section 4.6) is a special kind of Self-Pluggable Object that is suitable in cases when the object can be plugged in different application contexts (elements). The Multi-Context Self-Pluggable Object pattern is suitable for improving the *menu context* scenario described in Section 4.1.

An Abstract Self-Pluggable Object (Section 4.7) is a module suitable for structuring a set of related Self-Pluggable Objects, so that the behavior that plugs in those objects can be reused. It applies Factory Method (Gamma *et al.*, 1995) and can be implemented using the Template Advice idiom (Hanenberg *et al.*, 2003).

A Self-Pluggable Type Hierarchy (Section 4.8) is an alternative solution to structure a set of related Self-Pluggable Objects, which merges the implementation of types and the plugging of objects in the applications.

The patterns Abstract Self-Pluggable Object and Self-Pluggable Type Hierarchy are two alternatives that are suitable for solving a design problem that can emerge from applying either Self-Pluggable Object or Multi-Context Self-Pluggable Object.

Finally, an Association Object (Section 4.9) enables associations between Self-Pluggable Objects to be defined. This pattern is suitable for improving the *associating actions* scenario.

The examples of applying the patterns are given in Java, using AspectJ as the AOP language. Although the patterns were only experienced in AspectJ, they are not necessarily specific to it. An AOP language for a base object-oriented language, that features method execution pointcuts, abstract aspects, and abstract pointcuts, should be suitable for implementing the patterns. For instance, the patterns should be applicable to AspectC++ (Spinczyk *et al.*, 2002), which is the AspectJ counterpart for C++.

In the figures that illustrate the solutions, the framework modules are always represented in gray, whereas the white classes represent application modules. Aspects are depicted with a class with stereotype ≪`aspect`≫. Pointcuts and advices are represented in the method's placeholder using the stereotypes ≪pointcut≫ and ≪advice≫, accordingly. Stereotyped dependencies represent pointcut definitions, where the stereotype name represents the pointcut name.

Each pattern is identified by a *name*, which relates to the solution that the pattern proposes. The pattern descriptions are composed of the following sections:

- *Name.* A name that relates to the solution of the pattern.
- *Context.* This subsection locates the pattern in the context of other patterns, and describes the situation where the pattern is applicable.
- *Problem.* This subsection explicitly indicates the problem that the pattern is addressing by means of a question.
- *Forces.* This subsection contains a list of issues that discuss trade-offs and hypothetical benefits that lead to the proposed solution.
- *Solution.* This subsection describes the proposed solution, illustrated with a figure.
- *Example.* This subsection gives an example of an instance of the pattern by presenting code for the example framework.
- *Resulting context.* This subsection describes what is implied after applying the proposed solution.
- *Known uses.* This subsection gives examples where this pattern occurs, namely by referring to hot spots of the frameworks JHotDraw (SourceForge, 2006) and Eclipse RCP (McAffer & Lemieux, 2005).
- *Related patterns.* This subsection relates the pattern with the other patterns, describing possible combinations or alternatives.

4.3 Template Pointcut: an AspectJ Idiom

This section presents an AspectJ idiom referred to as Template Pointcut. Its name results from an analogy with the Template Method pattern. In a Template Method we have a partially implemented method which uses abstract methods that are given by subclasses. In the case of a Template Pointcut, we have a partially defined pointcut within an aspect module that uses abstract pointcuts that are given by the subaspects.

Context

An aspect module transforms (by *weaving*) other modules, which can be either classes or other aspect modules. A reusable abstract aspect is a module from which other aspects inherit (the subaspects), reusing its implementation. The scope of applicability of a reusable aspect may be restricted to a certain kind of base modules. The advantage of doing so is that the reusable aspect may assume certain characteristics of the modules which are going to be transformed. For instance, the reusable aspect may be applicable to all subclasses of a certain class, and therefore, the common inherited methods may be safely used by the aspect.

Problem

How to implement a reusable abstract aspect so that its advice can only take effect in a partially defined set of join points, while being able to generalize the commonalities between those join points?

Forces

- The information factored out to the reusable aspect should be maximized.
- The more "black-box" the reusable aspect is, the better.
- The simpler the pointcut definitions in the subaspects are, the better.

- The less one needs to know about the modules that an aspect transforms, the better.

Solution

Implement an abstract aspect containing a COMPOSITE POINTCUT (the *template*) that is defined as the intersection of certain join points with another ABSTRACT POINTCUT (the *hook*). The advice takes effect on the TEMPLATE POINTCUT (Figure 4.3). Subaspects of the abstract aspect have to define the hook pointcut.

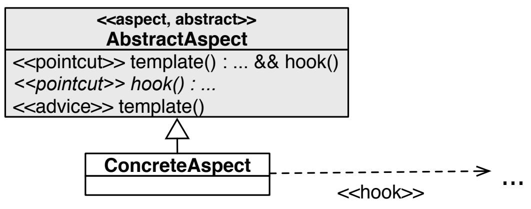


Figure 4.3: TEMPLATE POINTCUT idiom.

Example

Consider a reusable aspect that can be used to transform classes that inherit from the following abstract class.

```
public abstract class AbstractClass {
   /* ... */
   public String method();
}
```

The following is a reusable aspect with a TEMPLATE POINTCUT defined as the intersection between the execution of method() within subclasses of AbstractAspect and a hook class, which is to be given by the abstract pointcut hook(). The definition of hook() is intended to match a particular subclass of AbstractAspect, which the aspect will transform.

```
public abstract aspect AbstractAspect {
   private pointcut template() :
      within(AbstractClass+) && hook() && execution(String method());

   protected abstract pointcut hook();

   after() returning(String s) : template() {
      /* do something, e.g. */
      System.out.println(s);
   }
}
```

Although the pointcut template() is declared separately, it could be incorporated directly in the advice. Assuming the existence of a subclass of AbstractClass named SomeClass, the code below shows how the aspect could be used for activating the transformation of SomeClass.

```
public aspect ConcreteAspect extends AbstractAspect {
   protected pointcut hook() : target(SomeClass);
}
```

4.4 Composition Hook Method

Context

Template Method is an elementary and common pattern for enabling framework specialization, where adaptation is achieved by subclassing. The role of the hook methods that have to be overridden is often to plug objects into the application.

Problem

How to define hook methods for the purpose of enabling object plugging, so that they are intuitive to use?

Forces

- Reuse interfaces should be as simple as possible. By reading a hook method signature, it should be intuitive what the method has to do and how.

- The fewer framework methods one has to know for building an application, the better.

Solution

Define hook methods that expose in their parameters objects that are instantiated by the framework. These exposed objects are accessed by applications for composing other objects. The intent of a Composition Hook Method (Figure 4.4) is intuitively given by the method signature, while the way to plug objects into the exposed object is given by its interface.

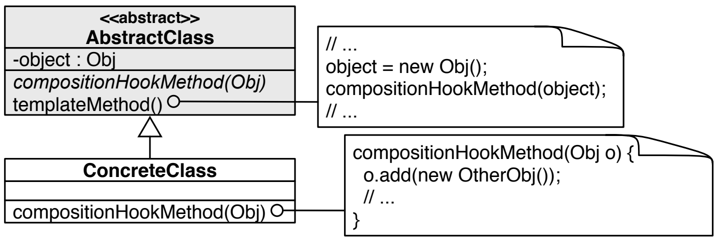


Figure 4.4: Composition Hook Method pattern.

Example

Considering the example framework, the abstract class AbstractApplication could be something like shown below. The class constructor is the template method, and there are two Composition Hook Methods for plugging in actions and menus.

```
public abstract class AbstractApplication {
  private ActionBar a_bar;
  private MenuBar m_bar;

  protected AbstractApplication() {
    a_bar = new ActionBar();
    createActions(a_bar);
    m_bar = new MenuBar();
    createMenus(m_bar);
  }
  protected abstract void createActions(ActionBar a_bar);
  protected abstract void createMenus(MenuBar m_bar);

  /* ... */
}
```

Resulting Context

- There is no need for additional methods whose purpose is to compose the object, which are done through the interface of the exposed objects.
- The way how to use hook methods is intuitive by reading their signature.

Known Uses

The main class of a JHotDraw-based application has to override Composition Hook Methods for plugging in *menus* and the *tools* that create the figures. A *viewpart* of an application based on Eclipse RCP has a Composition Hook Method for plugging GUI elements.

Related Patterns

The reader may indeed find a Template Method and this pattern very alike. However, the purpose of a Template Method is more generic, and the hook methods may have purposes other than enabling object composition.

By overriding a Composition Hook Method the variation is achieved through the exposed object. The type of objects that can be composed in the exposed objects is known, and there are methods in the objects' interface specifically for that purpose. Therefore, the body of an overridden Composition Hook Method is predictable with regard to the method invocations on the exposed object. For instance, the only purpose of the exposed object of type ActionBar in the given example is to perform register() calls with objects of type Action as arguments. All the implementations of this Composition Hook Method will be similar across framework-based applications.

A Composition Hook Method may become hidden from application developers, so that they will not need to deal with it when building an application. In order to do so, a Self-Pluggable Object is capable of dismissing the need of overriding the Composition Hook Method.

4.5 Self-Pluggable Object

Context

Framework classes have Composition Hook Methods.

Problem

How to eliminate the need for application developers to override a Composition Hook Method when implementing an application?

Forces

- Usually the type of the objects we want to plug in is easy to find. Finding the way to plug in the objects is the most difficult part, since the framework user has to understand the interface of the object where plugging takes place.
- The fewer hook methods application developers have to know and override, the better.
- The lack of documentation may cause application developers to plug objects in the wrong locations, resulting in incorrect uses of the framework.

Solution

Develop an abstract aspect that encapsulates the behavior that creates and plugs the object within a Composition Hook Method. Use a Template Pointcut where the fixed part defines the Composition Hook Method and the variable part (hook) is intended to match a subclass of the template class that owns that method. Application developers use a SELF-PLUGGABLE OBJECT (Figure 4.5) by extending the aspect and defining the hook pointcut on the desired context.

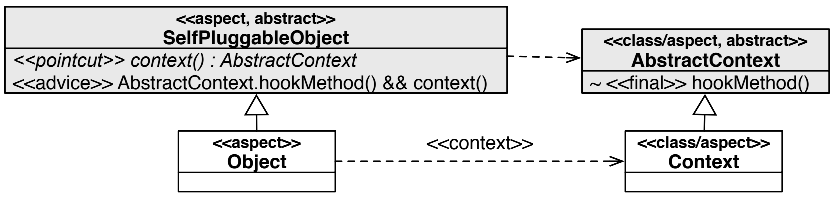


Figure 4.5: SELF-PLUGGABLE OBJECT pattern.

Example

This pattern is illustrated by presenting a solution for improving the *plugging menus* scenario given in Section 4.1. The COMPOSITION HOOK METHOD is AbstractApplication.createMenus(). Since this method will not need to be overridden by applications, it may have reduced visibility and be locked for overriding, as shown below.

```
public abstract class AbstractApplication {
  /* ... */
  final void createMenus(MenuBar m_bar) {   }
}
```

The following is a reusable aspect for plugging menus. The menu name is given in the constructor, while the menu is created by createMenu() using that name. The hook pointcut is application(). The advice creates the menu and plugs it into the MenuBar object parameter of createMenus(..). Using an independent method for creating the menu facilitates the collaboration with other aspects.

```
public abstract aspect Menu {
   private String name;

   public Menu(String name) {
      this.name = name;
   }
```

```
    protected abstract pointcut application();

    after(MenuBar mb) :
        within(AbstractApplication+) && application() &&
        execution(void createMenus(MenuBar)) && args(mb) {
            mb.add(createMenu());
        }

    IMenu createMenu() {
        return new MenuImpl(name);
    }
}
```

The main class of an application (subclass of AbstractApplication) does not need to override createMenu(..). ExampleApplication would be given like shown below.

```
public class ExampleApplication extends AbstractApplication {

}
```

In order to plug in a menu, a subaspect of Menu has to be defined. The following is an example of how to plug in a menu ("Menu1") in ExampleApplication.

```
public aspect Menu1 extends Menu {
    public Menu1() {
        super("Menu1");
    }

    protected pointcut application() : target(ExampleApplication);
}
```

In case the order of multiple Self-Pluggable Objects of the same type is relevant, precedences have to be used to explicitly declare the order in which the several objects are plugged in. The following example shows how it could be declared that Menu1 is to be plugged in before MenuX.

```
public aspect MenuOrder {
    declare precedence: MenuX, Menu1;
}
```

The precedence declaration may be given in an independent module as shown, but it can also be given together with the other modules.

Resulting Context

- Application developers no longer have to deal with the Composition Hook Method. Instead, they implement an independent aspect, which defines the hook pointcut.
- Objects can be plugged into other objects incrementally, without the need to modify, inspect, or understand, code related to the object where composition takes place.
- Changing the context where the object is composed can be done only by changing the hook pointcut definition.
- Framework-based applications are adaptable without the need of understanding or modifying source code. Removing a Self-Pluggable Object can simply be done by recompiling the application without its module (e.g. deactivating `Menu1` as a compilation unit).

Known Uses

Self-Pluggable Objects in JHotDraw can plug *menus*, *tools*, and *undo* on tools. Self-Pluggable Objects in Eclipse RCP can plug the *toolbar*, *perspectives*, and *viewparts* (an application can have several viewparts, which are organized in different perspectives).

Related Patterns

A Self-Pluggable Object can be plugged inro another Self-Pluggable Object. This can be done by intercepting the creation of objects in order to plug other objects into them, or by completing Composition Hook Methods, which Self-Pluggable Objects may have. A Self-Pluggable Object may be a Multi-Context Self-Pluggable Object if the object can be plugged into different application contexts. A Self-Pluggable Object may be based on an Abstract Self-Pluggable Object if there are multiple subtypes of the pluggable object. A Self-Pluggable Type Hierarchy merges the type implementations with their abstract composition (i.e. plugging). An Association Object enables the establishment of associations between Self-Pluggable Objects.

4.6 Multi-Context Self-Pluggable Object

Context

A Self-Pluggable Object is an object that plugs itself in a certain application context. However, there are objects which can be plugged into different application contexts.

Problem

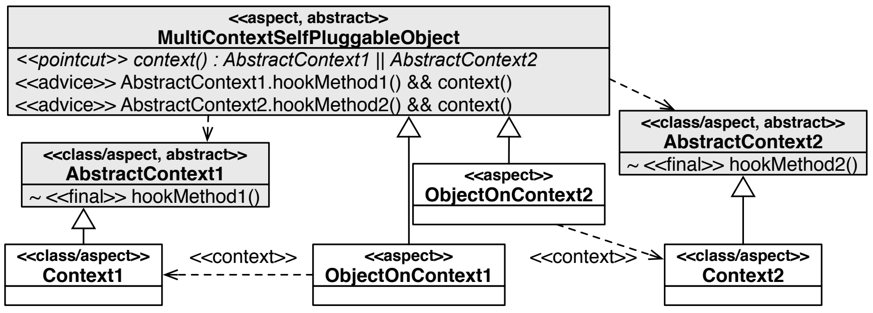
How to develop a Self-Pluggable Object that can be plugged into more than one application context?

Forces

- It is appealing to have everything that is possible to do with an object represented in a single module. By knowing that module, an application developer knows all that can be done with the object.
- If we would have a Self-Pluggable Object for each application context, there would be multiple modules for addressing a single concept.

Solution

Develop an aspect similar to a Self-Pluggable Object, which has one advice for each Composition Hook Method related with an application context. The hook pointcut is used in the different advices. When using the Multi-Context Self-Pluggable Object (Figure 4.6), the context is given by the module that is matched by the hook pointcut, implying that only the advice related to that application context will take effect.



<<aspect, abstract>>
MultiContextSelfPluggableObject
<<pointcut>> context() : AbstractContext1 || AbstractContext2
<<advice>> AbstractContext1.hookMethod1() && context()
<<advice>> AbstractContext2.hookMethod2() && context()
<<class/aspect, abstract>>
AbstractContext1
~ <<final>> hookMethod1()
<<class/aspect, abstract>>
AbstractContext2
~ <<final>> hookMethod2()
<<aspect>>
ObjectOnContext2
<<class/aspect>>
Context1
<<context>>
<<aspect>>
ObjectOnContext1
<<context>>
<<class/aspect>>
Context2

Figure 4.6: Multi-Context Self-Pluggable Object pattern.

Example

This pattern is illustrated by evolving the previous example, while addressing the improvement of the *menu context* scenario given in Section 4.1. Besides the application, a (sub-)menu can also be composed in another menu. Therefore, a menu can be used in more than one application context. The following is a new version of Menu, containing two advices. The first is like in the previous example, while the second is for addressing the composition of menus and sub-menus.

```
public abstract aspect Menu {

   /* to match an extension of either AbstractApplication or Menu */
   protected pointcut context();

   after(MenuBar mb) :
      within(AbstractApplication+) && context() &&
      execution(void createMenus(MenuBar)) && args(mb) {
         mb.add(createMenu());
      }

   after() returning(IMenu m) :
      within(Menu+) && context() &&
      execution(IMenu createMenu()) {
         m.addSubMenu(createMenu());
```

```
        }
    /* ... */
}
```

The following module would plug the menu "Menu1" in ExampleApplication (very similar to the example given previously).

```
public aspect Menu1 extends Menu {
    public Menu1() {
        super("Menu1");
    }

    protected pointcut context() : target(ExampleApplication);
}
```

The following module would plug the menu "Menu2" in the "Menu1".

```
public aspect Menu2 extends Menu {
    public Menu2() {
        super("Menu2");
    }

    protected pointcut context() : target(Menu1);
}
```

Resulting Context

- Everything that can be done in an application with an object is achieved through the same module.
- Changing the context where the object is composed, including different context types, can be done just by changing the hook pointcut definition in the object's module.

Known Uses

A Multi-Context Self-Pluggable Object in Eclipse RCP can plug *menus*, whose context may be (a) the application menu bar (conventional menu), (b) a certain *viewpart* (only shown in there), and (c) a certain *viewer* (appears as a pop-up menu).

Related Patterns

A Multi-Context Self-Pluggable Object is a special kind of Self-Pluggable Object.

4.7 Abstract Self-Pluggable Object

Context

Objects are plugged using Self-Pluggable Objects. A common case in frameworks is that objects of a certain type (e.g. represented by an interface) may be plugged in an application, and therefore, several subtypes of that type can be plugged in the same way.

Problem

When having a hierarchy of types whose objects can be plugged in an application, if we would have a Self-Pluggable Object for each one, there would exist duplicated code, given that all the objects are plugged in the same way. How to generalize the common behavior that is necessary to plug objects of a certain type?

Forces

- Code reuse should be maximized.
- A Self-Pluggable Type Hierarchy is also suitable to structure Self-Pluggable Objects, but this option is not always viable.

Solution

Develop an aspect similar to a Self-Pluggable Object, but with a Template Advice where the creation of the object to be plugged is done by a Factory Method (abstract method). This Abstract Self-Pluggable Object (Figure 4.7) should not be visible to applications. Develop one abstract aspect inheriting from it for each type to be plugged, where the implementation of the Factory Method returns the proper object. If application-specific objects of that type are allowed to be plugged, develop also an abstract aspect that implements the type but which does not implement the methods of the type, so that they can be given in application modules. An application developer may use one of the visible aspects by extending it and defining the hook pointcut. In case the application-specific type is intended to be implemented, the application developer extends the aspect for that purpose, and in addition to the hook pointcut definition, the type's methods have to be implemented.

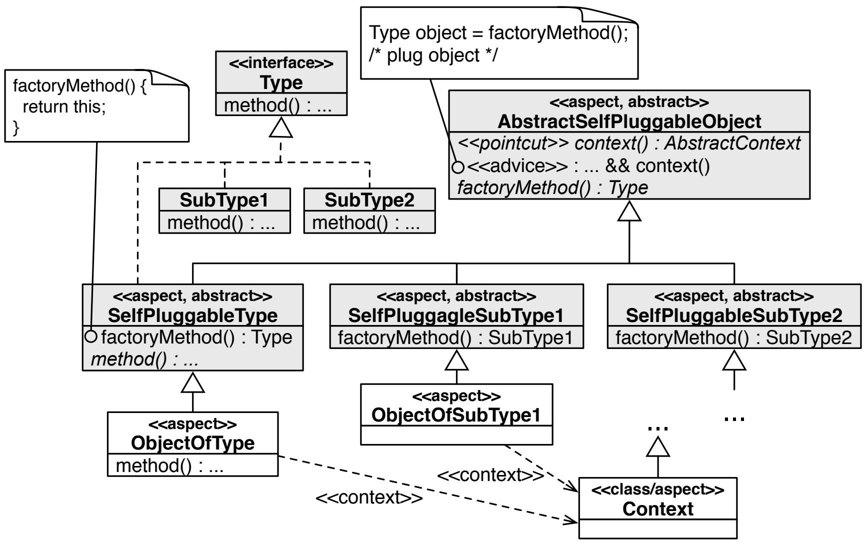


Figure 4.7: Abstract Self-Pluggable Object pattern.

Example

This pattern is illustrated with the plugging of *actions* in the example framework. The case of actions is very similar to the *plugging menus* scenario described in Section 4.1. Applications may include actions by plugging objects of type IAction. The following is an Abstract Self-Pluggable Object for this purpose. Except for the Template Advice, the solution is analogous to a Self-Pluggable Object.

```
abstract aspect AbstractAction {
   protected abstract pointcut application();

   after(ActionBar ab) :
      within(AbstractApplication+) && application() &&
      execution(void createActions(ActionBar)) && args(ab) {
         ab.register(createAction());
      }

   protected abstract IAction createAction();
}
```

The above aspect can be extended by SELF-PLUGGABLE OBJECTS, which can be used by application developers. The following is an example SELF-PLUGGABLE OBJECT, based on the ABSTRACT SELF-PLUGGABLE OBJECT, for addressing the *exit action*. The aspect overrides createAction() for returning an instance of the framework class ExitAction.

```
public abstract aspect Exit extends AbstractAction {
   protected IAction createAction() {
      return new ExitAction();
   }
}
```

The following module illustrates how Exit could be used, by plugging the exit action in ExampleApplication.

```
public aspect ExitOnExample extends Exit {
   protected pointcut application() : target(ExampleApplication);
}
```

The aspect that allows the plugging of application-specific actions could be implemented like shown below. The aspect implements IAction, but it is up to applications to implement the interface methods (run() in this case).

```
public abstract aspect Action extends AbstractAction
   implements IAction {

   public abstract run();

   protected IAction createAction() {
        return this;
    }
}
```

The following module illustrates how Action could be used. In addition to the hook pointcut definition, the method implementation is given.

```
public aspect ExampleAction extends Action {
    public void run() {
        /* application-specific action implementation */
    }

    protected pointcut application() : target(ExampleApplication);
}
```

Resulting Context

- The plugging of objects of a certain type is generalized. Support for new types can be added simply by developing an aspect module that overrides the FACTORY METHOD (e.g. as in Exit).

- The code of the modules that extend the ABSTRACT SELF-PLUGGABLE OBJECT still has some redundancy given that the implementation of the FACTORY METHODS across the several modules is very similar (only the name of the class changes).

- The number of subaspects grows along with the number of framework-provided type implementations, implying one more framework module for each one (i.e. the aspect).

Known Uses

In Eclipse RCP, an ABSTRACT SELF-PLUGGABLE OBJECT can generalize the plugging of *actions*, which may be either chosen from a set of framework-provided actions or implemented by applications. In JHotDraw there is an analogous case for plugging *commands*.

Related Patterns

An ABSTRACT SELF-PLUGGABLE OBJECT serves the purpose of structuring a set of related SELF-PLUGGABLE OBJECTS, and consists of an alternative to a SELF-PLUGGABLE TYPE HIERARCHY.

4.8 Self-Pluggable Type Hierarchy

Context

An ABSTRACT SELF-PLUGGABLE OBJECT is capable of generalizing the plugging of objects of a certain type, implying that there will exist an aspect for each type. All these subaspects are similar and only differ in the object instance returned by the FACTORY METHOD.

Problem

How to avoid having several similar subaspects when structuring SELF-PLUGGABLE OBJECTS by means of an ABSTRACT SELF-PLUGGABLE OBJECT?

Forces

- In solutions based on an ABSTRACT SELF-PLUGGABLE OBJECT, the number of subaspects grows along with the number of framework-provided type implementations. The disadvantage is that the solution implies one SELF-PLUGGABLE OBJECT for each pluggable type.

- The fewer framework modules there are, the better.

- Similar or redundant code across related modules should be avoided.

Solution

Merge the implementation of a type hierarchy of default components with the SELF-PLUGGABLE OBJECTS that implement the composition of objects of that type. In order to do so, develop an aspect similar to a SELF-PLUGGABLE OBJECT that is of the top-most type of the hierarchy (i.e. it declares that it implements that type), while it not implements the type's methods. Develop a subaspect for each subtype, where the type methods are implemented. These aspects may represent partial type implementations by implementing a subset of the type methods, while leaving the remaining methods to applications. Application developers can use the appropriate member of the SELF-PLUGGABLE TYPE HIERARCHY (Figure 4.8) that implements the type they wish to use. If an application has to include its own implementation of the type, it extends the top-most aspect.

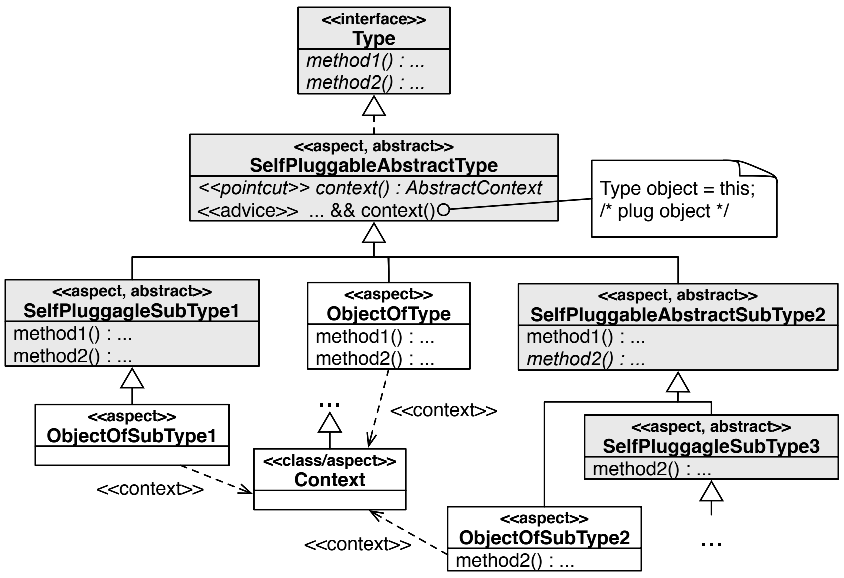


Figure 4.8: SELF-PLUGGABLE TYPE HIERARCHY pattern.

Example

This pattern is illustrated with the same case of the *actions* in the example framework, as it consists of an alternative solution to the one given in Section 4.7.

The following is a new version of Action that can be used in the same way by application developers (exemplified in Section 4.7). The aspect is of type IAction, and registers itself as an action.

```
public abstract aspect Action implements IAction {
   protected abstract pointcut application();

   after(ActionBar ab) :
      within(AbstractApplication+) && application() &&
      execution(void createActions(ActionBar)) && args(ab) {
         ab.register(this);
      }

   public abstract void run();
}
```

The following is a new version of Exit that can be used in the same way by application developers (as given in Section 4.7).

```
public abstract aspect Exit extends Action {
  public void run() {
    /* the exit action implementation */
  }
}
```

Resulting Context

- The solution involves fewer framework modules when comparing with a solution based on an ABSTRACT SELF-PLUGGABLE OBJECT, while it is more elegant given there is no similar or redundant code.
- The types addressed by the SELF-PLUGGABLE TYPE HIERARCHY cannot be instantiated independently.

Known Uses

In JHotDraw, a SELF-PLUGGABLE TYPE HIERARCHY is capable of organizing the framework-provided *figures* and *connection figures*. In order to use an application-specific figure, application developers may extend a member of the hierarchy, which may be the top element for implementing a completely new figure, or a lower one in case the figure is intended to be based on an existing one.

Related Patterns

If merging the implementation of the types with Self-Pluggable Objects is not possible due to some constraint, a solution based on an Abstract Self-Pluggable Object can be used instead.

4.9 Association Object

Context

Objects are plugged into an application using Self-Pluggable Objects or Multi-Context Self-Pluggable Objects. The objects plugged into an application may need to have other associations between them.

Problem

The plugged objects are not visible to application developers, so that they can define the associations. How to establish an association between two Self-Pluggable Objects?

Forces

- Having the possibility of managing object associations independently is advantageous because application features relying on associations may be plugged and unplugged without modifying other modules.

- Given that the objects are handled by Self-Pluggable Objects, it makes sense to define associations in terms of these modules.

Solution

Develop an abstract aspect, which when made concrete encapsulates an association between two objects — an Association Object (Figure 4.9). Use two Template Pointcuts to capture the creation of the objects, each one with its own advice. One advice captures and stores a reference to one of the objects, while the other advice uses that reference to establish the association with its captured object. Application developers can establish an association by defining the hook pointcuts on the modules representing the two objects to be associated. An Association Object can be defined in terms of an Abstract Self-Pluggable Object or the top-level aspect of a Self-Pluggable Type Hierarchy. This enables that the association can be established between any object whose type is a subtype of the type addressed by either the Abstract Self-Pluggable Object or the Self-Pluggable Type Hierarchy.

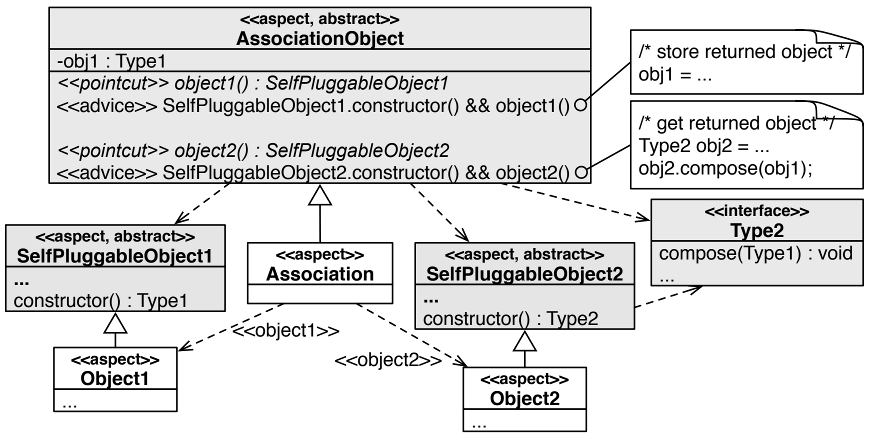


Figure 4.9: Association Object pattern.

Example

This pattern is illustrated by presenting a solution for improving the *associating actions* scenario given in Section 4.1. Below we present the aspect that enables the encapsulation of such associations, assuming the specialization aspect Action from Section 4.8 and the specialization aspect Menu from Section 4.6. The first advice captures the instantiation of a subaspect of Action, which is itself an object of type IAction, while the second advice includes the captured action in a menu captured from the execution of createMenu() within a subaspect of Menu.

```
public abstract aspect MenuAction {
   protected abstract pointcut action();
   protected abstract pointcut menu();
   private IAction a;

   after(IAction a) :
      within(Action+) && action() &&
      execution(Action.new(..)) && this(a) {
         this.a = a;
      }

   after() returning(IMenu m) :
      within(Menu+) && menu() &&
      execution(IMenu createMenu()) {
         m.addAction(a);
      }
}
```

Assuming the SELF-PLUGGABLE OBJECTS Menu1 and ExitOnExample given previously, the following aspect implements the association that places the *exit action* on "Menu1".

```
public aspect ExitOnMenu1 extends MenuAction {
   protected pointcut action() : target(ExitOnExample);
   protected pointcut menu() : target(Menu1);
}
```

Resulting Context

- Associations can be encapsulated and managed independently.
- In order to establish an association there is no need to understand the context where the objects are plugged nor any details about their type implementation.

Known Uses

In JHotDraw an Association Object is capable of defining the valid source and target *figures* which a *connection figure* may connect. However, the solution is a bit different than the one in the example, since the valid connections are given by overriding a hook method of the connection figure. In Eclipse RCP, Association Objects are capable of associating *actions* with the *toolbar*, the *actions* with the *menus*, or *viewparts* with the *perspectives*.

Related Patterns

An Association Object may be adaptable by having Composition Hook Methods, which in turn can be completed by Self-Pluggable Objects.

4.10 Example Framework Revisited

This section revisits the example framework, taking into account the Modular Hot Spots pattern language given throughout sections 4.5-4.9. Figure 4.10 depicts the new reuse interface after applying the patterns. We can see the several abstract modules (gray) and their abstract pointcuts. Regarding the *menus*, the example of Section 4.6 is considered (Multi-Context Self-Pluggable Object), instead of the one given in Section 4.5. Regarding the *actions*, the examples of Section 4.8 are considered (Self-Pluggable Type Hierarchy), instead of the ones of Section 4.7.

Figure 4.10 also depicts the framework-based application based on Modular Hot Spots that was given throughout the patterns. We can see the several application aspects (inheriting from the specialization aspects) and their pointcut definitions. The following points briefly compare this solution with the conventional one given in Section 4.1.

1. Each of the application concepts (i.e. *application*, *menu*, *action*, *exit*, and *menu action*) can be used incrementally, where each concept instance is implemented in an independent module. Throughout the pattern examples a framework-based application was given in the modules ExampleApplication

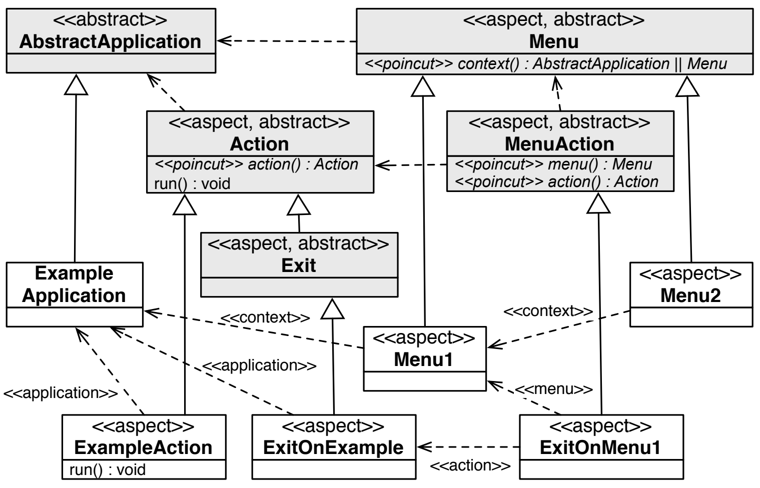


Figure 4.10: Modular Hot Spots for the example framework.

(Section 4.5), Menu1 and Menu2 (Section 4.6), ExitOnExample and ExampleAction (Section 4.7), and ExitOnMenu1 (Section 4.9).

2. Without source code modification, the application may be compiled with subsets of the modules enumerated in (1), obtaining variants of the application. This issue is particularly important in the context of *software product-lines*. For instance, one could have a variant of the application without the "Menu2", just by not including that module in the compilation.

3. Application features can be removed without understanding source code, as far as one knows which application elements the modules are representing (a fairly basic information that is easy to maintain). This issue facilitates the *maintenance* and *reengineering* of framework-based applications. For instance, suppose that the application implemented by the modules enumerated in (1) needs to be changed for a new version without the ExampleAction. If the task is given to a programmer that was not the one who developed the application in the first place, his or her task becomes facilitated, given that only that module has to be identified and removed, while no understanding of the existing code is necessary.

4. The associations between menus and actions can be independently and non-invasively defined.

5. The two hook methods of AbstractApplication, plus the two methods of Menu, of the conventional reuse interface, no longer have to be dealt with by application developers. Instead, there are pointcuts that assume their role. The advantages of the latter is that compositions can take place without modifications and inspection of the target modules.

6. The two classes MenuBar and ActionBar are no longer relevant for the application developer.

The items (1), (2), and (3) are related with the improvement of the *plugging menus* and *menu context* scenarios given in Section 4.1. Item (4) is related with the improvement of the *associating actions* scenario also given in Section 4.1.

4.11 Discussion

Modular Hot Spots, i.e. a set of specialization aspects, can form a black-box reuse interface with a higher level of abstraction than a conventional black-box reuse interface. Black-box frameworks are pointed out as adequate for having an accompanying Visual Builder (Roberts & Johnson, 1997) for generating framework-based applications from high-level domain-specific descriptions. Such Visual Builders, referred to as F-DSLs in this dissertation, can be developed more easily if having Modular Hot Spots, given that application code resembles more closely the concepts and relationships of a given domain.

A Visual Builder requires a definition of a language of domain-specific concepts. Chapter 5 presents an approach for defining those concepts as a set of specialization aspects. The solutions for such specialization aspects are closely related to the patterns of Modular Hot Spots, in the sense that each pattern is associated with one modeling construct that can be used in the definition of the language of domain-specific concepts.

Chapter 5

Domain-Oriented Reuse Interfaces

This chapter addresses the concept of a domain-oriented reuse interface (DORI) and its realization using framework specialization aspects (SAs). Section 5.1 gives an overview of the approach and outlines the role of SAs in DORIs. Section 5.2 explains the concept of a DORI. Section 5.3 describes the modeling constructs that can be represented in a DORI and how to use them. Section 5.4 presents a DORI for an example framework, illustrating the several modeling constructs. Finally, Section 5.5 concludes the chapter by emphasizing the value of DORIs for developing F-DSLs.

5.1 Overview

For a given framework, a DORI is a reuse interface that is structured according to the *domain concepts* (or simply, concepts) involved in framework-based applications. The idea behind the words "domain-oriented" in the acronym DORI, is to emphasize that the reuse interface is intended to have a close relation with a *domain variability model* (Pohl *et al.*, 2005).

A domain variability model is suitable for defining the concepts of an F-DSL and can be given as a conceptual model. Application models described using an F-DSL are defined in terms of instances of the concepts of the domain variability model. The two major requirements of a DORI are the following:

1. The reuse interface unambiguously defines a domain variability model for the framework. This implies that from a DORI one can obtain a unique conceptual model that represents a domain variability model.

2. From an application model, i.e. an instance of the conceptual model of (1), application code based on the reuse interface can be obtained unambiguously, without having a transformation definition for each DORI. This implies that it is possible to have a generic procedure, applicable to every DORI, for generating application code from application models.

Requirement (1) enables the definition of concepts of an F-DSL to be in the framework itself, or more concretely, in its reuse interface. Requirement (2) enables one to have a generic transformation definition, which can be realized in a generic code generator, and thus, it avoids the need of developing specific code generators for each F-DSL. In this way, a framework with a DORI not only implements an abstract solution to a family of related problems, but it also implements an F-DSL for building applications of that family.

Figure 5.1 presents a use case diagram depicting the role of domain and application engineers in DORI-based development. The stereotype <<requires>> was used to denote that a use case requires the behavior of the other use case that

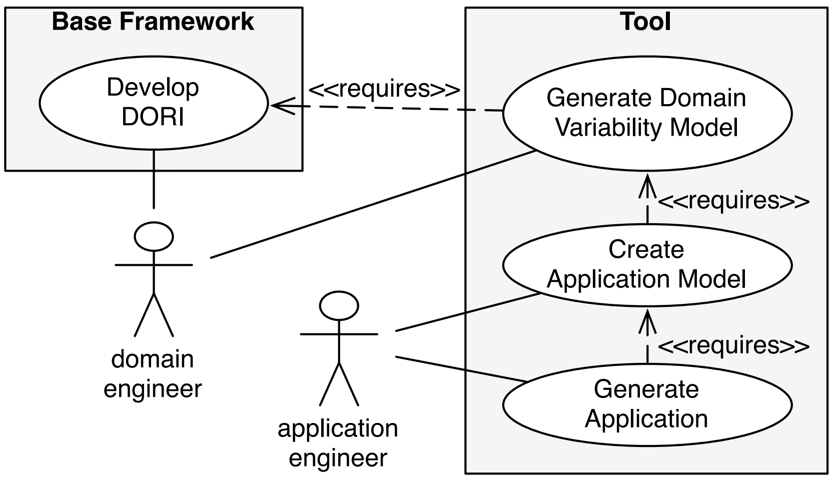


Figure 5.1: The role of domain and application engineers in DORI-based development.

it depends on. A domain engineer has the task of developing a DORI for a base framework. By using a generic tool, he or she can generate the domain variability model that is expressed in the DORI. On the other hand, an application engineer uses the tool for creating an application model, which instantiates the domain variability model. In turn, he or she uses the tool to generate the application from the application model.

Several hot spots are represented in the reuse interface of a framework. Each hot spot supports the instantiation of a concept in a framework-based application. In a conventional framework, the several hot spots overlap in framework classes (recall Figure 2.1). Moreover, different concepts are instantiated in framework-based applications using different adaptation mechanisms, such as inheritance, hook methods, or object composition. Due to the many-to-many mapping from concepts to framework classes and the non-uniform way of using framework concepts, requirements (1) and (2) cannot be easily satisfied by means of a conventional reuse interface.

As explained in Chapters 3-4, SAs are capable of expressing hot spots and have the following properties:

a. Each concept offered by the framework is able to be represented in a single and independent module — an SA. Hot spots are captured by means of SAs, providing a uniform reuse interface to develop framework-based applications.

b. Each concept instance is represented in a single and independent module based on an SA — an AA. Concepts are instantiated uniformly. A concept instance is represented by inheritance, parameterization is represented by invoking the super constructor, and association links between concept instances are represented by pointcut definitions to other AAs.

In an F-DSL, the concepts provided by the framework are typically first-class entities in the language. Property (a) enables a close relation between SAs and F-DSL concepts. Property (b) enables a close relation between AAs and instances of F-DSL concepts. Therefore, an SA is an adequate artifact to express a F-DSL concept, whereas an AA is an adequate artifact to express an instance of an F-DSL concept.

5.2 Concept

Figure 5.2 presents a conceptual model describing DORIs and DORI-based applications. This model contains classes in common with the conceptual model of Figure 3.3. Such classes relate to SAs and AAs, and are represented in gray.

A *DORI* expresses a *domain variability model* given as a conceptual model, which is composed of several *concepts*. A DORI is composed of several *DORI modules*. A DORI module bridges a concept of the domain variability model and an SA. Each concept is either *independent* or *dependent*. The former is a concept whose instances may exist by their own, whereas the latter is a concept whose instances may only exist as a part of another concept instance. An independent concept may be, for example, a stand-alone application, while a dependent concept may be a menu that is part of an application. An independent concept is represented in terms of an SA that is an *abstract class*, whereas a dependent concept is represented in terms of an SA that is an *abstract aspect*.

A dependent concept has to define one or more *composite associations*, each of which defining its parent (i.e. the concept that contains the dependent concept). A dependent concept may also define *directed associations* with other target concepts. Each of these *relationships* is represented in terms of an *abstract pointcut* that is part of the abstract aspect that represents the dependent concept.

A concept may have several *attributes*. Each attribute is represented in terms of a *parameter* of the SA that represents the concept. A concept may have a super concept. If the SA that represents the concept has a super SA, then the super concept is the concept represented by the super SA. A concept inherits the attributes and relationships of its super concept.

An *application model* describes a DORI-based application in terms of several *concept instances*, which are instances of the concepts of the domain variability model. Each concept instance is represented through an *application aspect* (AA). The AA may be either a *concrete class* or a *concrete aspect*, according to the SA from which it inherits. A concept instance may contain *association links* to other concept instances. Such association links are instances of the relationships between concepts. In fact, only the instances of dependent concepts may have

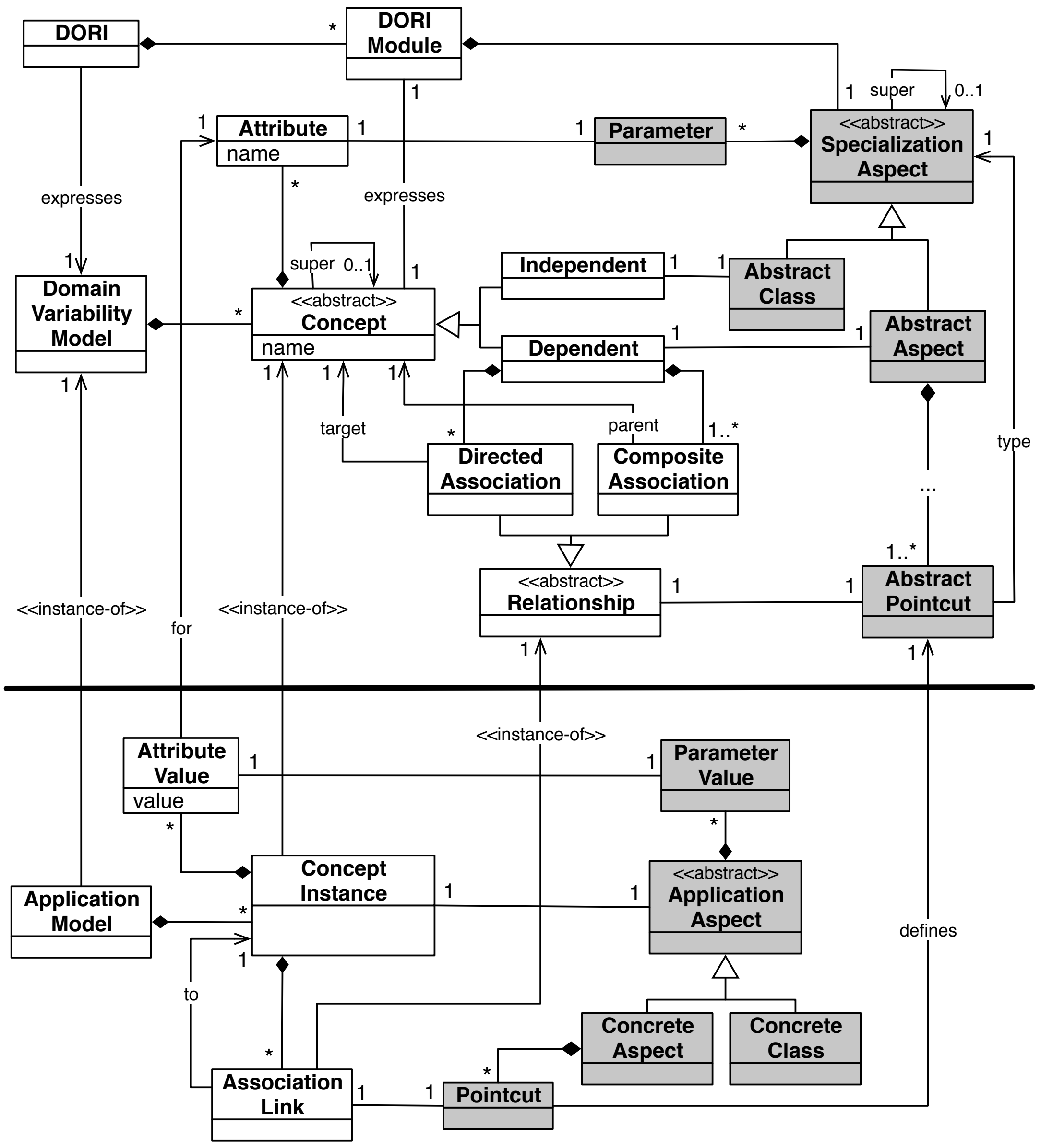


Figure 5.2: Conceptual model of DORIs (top) and DORI-based applications (bottom).

association links, but this constraint is not visible in the model for reducing cluttering. Concrete aspects contain *pointcuts* that define the *abstract pointcuts* of the SA. Each association link is represented in terms of one of these pointcuts. Finally, each concept instance contains one *attribute value* for each concept attribute. Each attribute value is represented in terms of a *parameter value* of the AA that represents the concept instance. The instance of a concept that has a super concept also has to contain attribute values and association links according to the attributes and relationships of the super concept.

The mapping between modeling elements (i.e. concepts, attributes, and relationships) and SAs (including their parameters and abstract pointcuts), has to be explicitly specified. The next section details a technical solution for defining this mapping, explaining how the modeling constructs can be used.

5.3 Expressing Modeling Constructs

The previous section presented the concept of DORI in terms of the relation between a domain variability model and SAs, and the relation between application models and AAs. This section presents a technical solution based on annotations for mapping concepts and SAs. In addition to the general modeling constructs given in the previous section, three special kinds of concepts for addressing integration of manual code with generated code are also introduced in this section.

Figure 5.2 described the basic conceptual modeling constructs that can be expressed in a DORI. Figure 5.3 describes all the supported modeling constructs with a conceptual model, including the constructs that were already introduced in Figure 5.2. This model includes the different kinds of concepts, while it omits details regarding the concepts being either independent or dependent. As described in the model of Figure 5.2, the specialization aspect kind (i.e. either abstract class or abstract aspect) determines if the concept is either independent or dependent.

An *abstract concept* is a concept that cannot be instantiated. A *public* concept is a concept whose instances are to be transformed into an application module that is intended to be used manually, e.g. a class that will be instantiated within another application. An *open concept* is a concept that is representing an *open variation point* (Gurp *et al.*, 2001). An open variation point is an adaptable part that framework-based applications may extend by providing arbitrary application code, which cannot be anticipated. In a DORI, instances of open concepts are adapted by implementing well-defined methods — the *open methods*. These

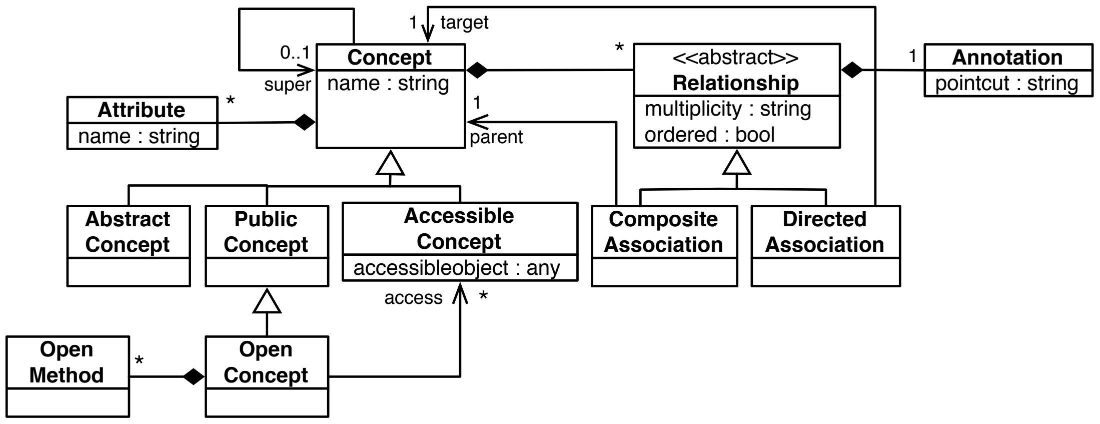


Figure 5.3: Modeling constructs that can be represented in a DORI.

methods are intended to be exposed to application developers in isolation. Since an application developer has to manipulate the modules, an open concept is a public concept. An *accessible concept* is a concept whose instances have an *accessible object* that may be accessed by instances of open concepts. These two kinds of concepts serve the purpose of realizing the integration of manual code and code that can be generated from models.

A relationship has a *multiplicity* associated with the *parent* or *target* concept (depending on the association kind), and it may be *ordered* or not. A relationship has an attached *annotation* that stores the name of the abstract *pointcut* which represents it.

5.3.1 Annotations

Figure 5.2 described how the concepts of a domain variability model relate to SAs. A possible technical solution for associating the concepts, their attributes and relationships, with the classes/aspects, their parameters and abstract pointcuts, is to annotate the SAs in order to define this mapping.

A regular concept is expressed by annotating an SA with `@Concept`. In addition, there are four other kinds of concepts that are expressed with their own annotation. An abstract concept is expressed using `@AbstractConcept`, a public concept is expressed using `@PublicConcept`, an accessible concept is expressed using @AccessibleConcept, and finally, an open concept is expressed using @OpenConcept. The name of the SA defines the name of the concept which the SA is representing.

An abstract class is considered to represent an independent concept, whereas an abstract aspect is considered to represent a dependent concept. An abstract aspect must have at least one @PartOf annotation on an abstract pointcut, for defining a composite association with the parent concept. The following code shows a sample DORI module representing a concept Child which has the concept Parent as its parent. Each instance of Parent has to contain one or more child instances of Child, given that the multiplicity is “1..*” (annotation attribute mult). Moreover, the child instances are ordered, given that the annotation attribute ordered is set to true.

```
@Concept
public abstract aspect Child {
  /* ... */

  @PartOf(concept="Parent", mult="1..*", ordered=true)
  protected abstract pointcut parent();
}
```

A child concept may define multiple parent concepts. In this case, there is a @PartOf annotation for each pointcut that represents a composite association with the parent concept.

A concept defined by a DORI module may have directed associations to other concepts by annotating an abstract pointcut with @Association. The following code shows the same abstract aspect of the previous example defining a directed association with multiplicity “1” to RelatedConcept. This implies that each instance of Child has to define one association link to an instance of RelatedConcept.

```
@Concept
public abstract aspect Child {
  /* ... */

  @Association(concept="RelatedConcept", mult="1")
  protected abstract pointcut concept();
}
```

When a DORI module defines an accessible concept, it must have an annotation @AccessibleObject on one of its attributes, for defining which is the accessible object. The object referenced by that attribute will be the accessible object associated with the concept instance. The following code shows a sample DORI module representing an accessible concept, defining the attribute entity as the accessible object.

```
@AccessibleConcept
public abstract aspect AccessibleConcept {
  @AccessibleObject
  Entity entity;

  /* ... */
}
```

A DORI module representing an open concept must have at least one of its abstract methods annotated with @OpenMethod, for defining the open methods. The following code shows a sample of a DORI module representing an open concept with the open method open().

```
@OpenConcept
public abstract aspect OpenConcept {
  @OpenMethod
  protected abstract void open();
  /* ... */
}
```

The attributes of a concept are represented by annotating the module's constructor with the annotation @Attributes. This implies that each parameter of the constructor is expressing an attribute of the concept. The following code shows a sample DORI module representing a concept with the attributes a and b.

```
@Concept
public abstract aspect Concept {
  @Attributes
  public Concept(String a, int b) {
    /* ... */
  }
  /* ... */
}
```

5.4 DORI Example

This section presents a DORI for an example framework, illustrating all modeling constructs detailed in Section 5.3.

5.4.1 Example Framework

For the purpose of giving examples of how to represent the several modeling constructs in a DORI, this section introduces an example GUI framework. The example is a fragment extracted from the Eclipse RCP framework, which was simplified for clarity of explanation.

5.4.1.1 Domain variability model

Figure 5.4 presents a domain variability model for the framework. An *application* has a certain *name* that appears as the window title. It may have several *actions*. An action may be framework-provided, as for instance the *maximize* and *exit* actions, or application-specific. In the latter case, they are referred to as *application actions* and a *name* has to be given to them. An application is considered an open concept, given that its behavior is arbitrary, whereas the exit action is considered an accessible concept because other actions may need to execute it. For instance, an application action could be a "save and quit" action, where all data is saved first and the application is terminated after. An application may have several *menus*, whose order is relevant because it defines how they appear

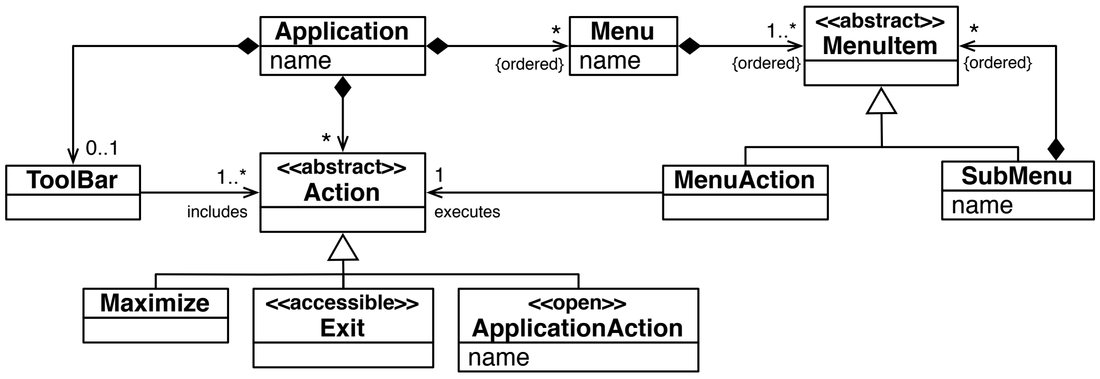
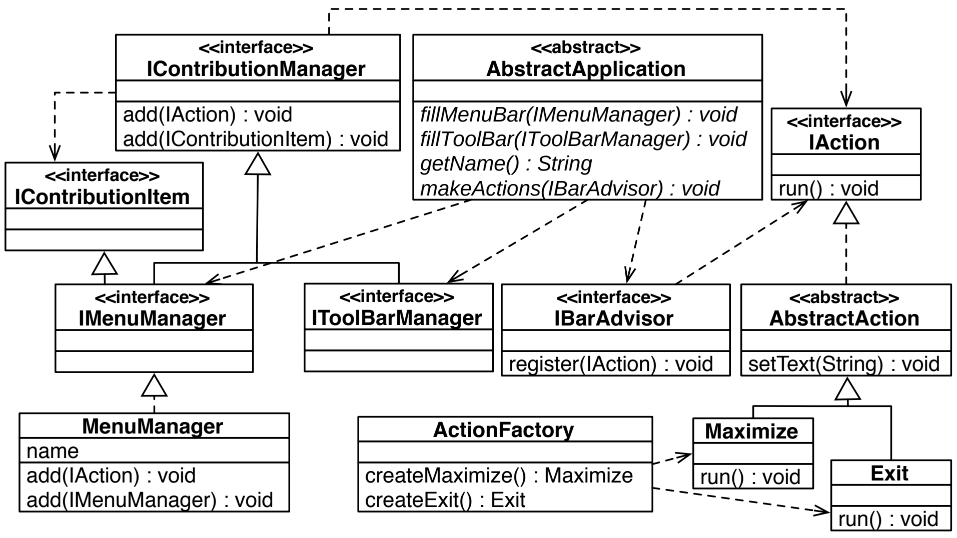


Figure 5.4: Domain variability model of the example framework.

on the menu bar. Each menu has a *name* and contains one or more *menu items*. A menu item can be either a *menu action* that executes a certain action, or a *sub menu* that has a *name* and may also contain several menu items. The menu items are also ordered. Finally, an application may have a toolbar, which may include one or more actions. The order of the actions in the toolbar is controlled by the framework.

5.4.1.2 Conventional reuse interface

Figure 5.5 presents the classes of the reuse interface of the example framework that are relevant with respect to the concepts included in Figure 5.4. The figure only shows the attributes and methods that are relevant in the context of the given concepts.



<<interface>> IContributionManager
add(IAction) : void
add(IContributionItem) : void
<<abstract>> AbstractApplication
fillMenuBar(IMenuManager) : void
fillToolBar(IToolBarManager) : void
getName() : String
makeActions(IBarAdvisor) : void
<<interface>> IAction
run() : void
<<interface>> IContributionItem
<<interface>> IMenuManager
<<interface>> IToolBarManager
<<interface>> IBarAdvisor
register(IAction) : void
<<abstract>> AbstractAction
setText(String) : void
MenuManager
name
add(IAction) : void
add(IMenuManager) : void
ActionFactory
createMaximize() : Maximize
createExit() : Exit
Maximize
run() : void
Exit
run() : void

Figure 5.5: Conventional reuse interface of the example framework.

The following list presents the necessary tasks for using the concepts in a framework-based application:

- An application has to extend the abstract class AbstractApplication and to implement the abstract methods. The method getName() defines the window name, makeActions() allows actions to be plugged through an object of type IBarAdvisor, fillMenuBar() allows menus to be plugged through an object of type IMenuManager, and fillToolBar() allows actions to be plugged in the tool bar through an object of type IToolBarManager.

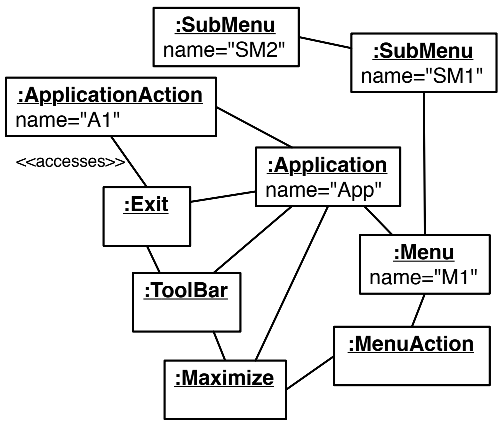
- An action is represented by the interface IAction. There is an abstract class AbstractAction from which all the classes implementing actions are likely to inherit from. Framework-based applications may define their own actions by extending AbstractAction. On the other hand, framework-provided actions can be used (for instance Maximize or Exit), and they should be created using the ActionFactory.

- A menu is represented by the interface IMenuManager and realized in the class MenuManager. A submenu is also represented by the same class. Menus may contain contribution items, which are represented by the interface IContributionItem, and IMenuManager is a contribution item itself. Menus may include actions. A menu can be filled using the method add().

- The application will have a tool bar if at least one action is plugged into the application, by invoking add() on the IToolBarManager object when implementing fillToolBar().

In order to use the given concepts in a framework-based application, an application developer has to deal with the given reuse interface. Notice that the classes contain much more detail than an application description using the actual domain terminology would require. For instance, the hook methods of AbstractApplication, the IBarAdvisor type, or the ActionFactory, are implementation-related and therefore they are solution space artifacts, rather than problem space artifacts that map directly to a domain concept. On the other hand, the classes may not explicitly represent certain domain concepts. For instance, the submenu does not have a dedicated class representing it. Moreover, the locations where application-specific functionality can be added (i.e. open variation points) are also not made explicit.

Figure 5.6 presents an object diagram describing an example application model, which is an instance of the domain variability model given in Figure 5.4.



```
public class A1 extends AbstractAction {
  private IAction exit;

  public A1(IAction exit) {
    setText("A1");
    this.exit = exit;
  }

  public void run() {
    // do something
    exit.run();
  }
}
```

```
public class App extends AbstractApplication {
  private static IAction action_max;
  private static IAction action_exit;
  private static IAction action_app;

  public String getName() {
    return "App";
  }

  public void makeActions(IBarAdvisor b) {
    action_max = ActionFactory.createMaximize();
    b.register(action_max);
    action_exit = ActionFactory.createExit();
    b.register(action_exit);
    action_app = new A1(action_exit);
    b.register(action_app);
  }

  public void fillMenuBar(IMenuManager m) {
    MenuManager menu_m1 = new MenuManager("M1");
    menu_m1.add(action_max);
    m.add(menu_m1);
    MenuManager submenu_sm1 = new MenuManager("SM1");
    menu_m1.add(submenu_sm1);
    MenuManager submenu_sm2 = new MenuManager("SM2");
    submenu_sm2.add(submenu_sm1);
  }

  public void fillToolBar(IToolBarManager t) {
    t.add(action_exit);
    t.add(action_max);
  }
}
```

Figure 5.6: Example application model and its realization.

The figure also shows the application code that implements the model in the modules App and A1, using the reuse interface given in Figure 5.5. The application action “A1” does something and uses the exit action to quit the application.

5.4.2 DORI Modules

This subsection illustrates the use of the several modeling constructs by presenting DORI modules for the given example framework. Each example covers a fragment of the given domain variability model (Figure 5.4) and a fragment of the conventional reuse interface (Figure 5.5). The AOP language used in the examples is AspectJ (Eclipse Foundation, 2007a). All the code examples are complete, except for both package and import declarations, which are omitted. When appropriate, the examples refer to the patterns given in Chapter 4.

Each modeling construct is illustrated with an example given in a figure that has the following graphical layout (see Figure 5.7):

1. The upper part presents DORI modules, together with a class diagram showing the fragment of the domain variability model that they represent. The white classes represent the concepts that actually defined by the DORI modules of the figure, whereas the classes that are shaded are not defined by the DORI modules of the figure. Such shaded classes represent concepts defined by DORI modules of earlier examples. All the classes that are presented can be unambiguously generated from the DORI modules using a tool.

2. The bottom part presents an object diagram showing a fragment of an example application model that is an instance of the class diagram of (1), together with the AAs that realize that model. Analogously to (1), when objects are shaded in the diagram it means that they were introduced in an earlier example, but their AAs are now shown in the figure. All the AAs that are presented can be unambiguously generated from the application model using a tool.

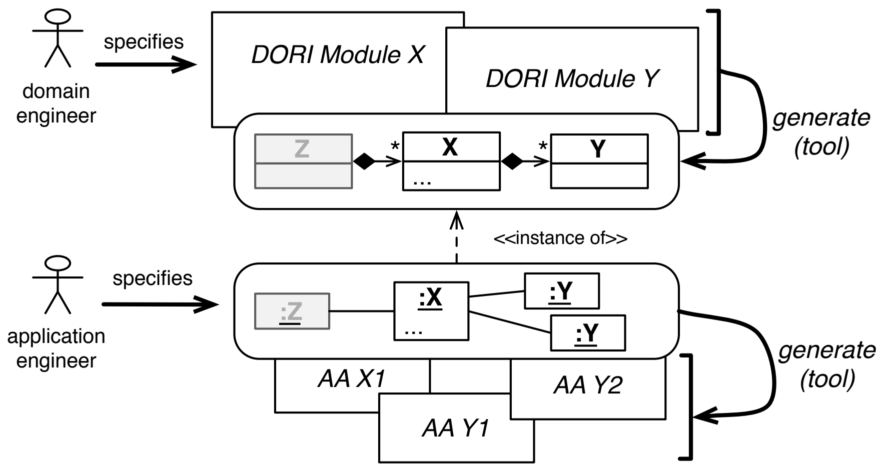


Figure 5.7: Graphical layout of the presentation of DORI Module examples.

5.4.2.1 Concepts and attributes

In order to illustrate concepts and attributes, consider the case of the independent concept *application*. As described, a subclass of AbstractApplication has to be provided in order to implement an application. According to the domain variability model, the concept of having an empty application only involves the application name. Therefore, other details, such as the hook methods, shall not be handled by application code. Figure 5.8 presents a DORI module that expresses the *application* concept. This concept is considered public since this module is the main class of the application, which can be used in other artifacts (e.g. a descriptor file). The module name defines the concept name (i.e. "Application"), while the annotated constructor defines the concept's attributes (i.e. "name" of type string).

In conventional reuse interfaces, abstract classes that are intended to be specialized by a framework-based application commonly have Composition Hook Methods (Section 4.4). In this case we have the methods makeActions(), fillMenuBar(), and fillToolBar(), which conform to this pattern.

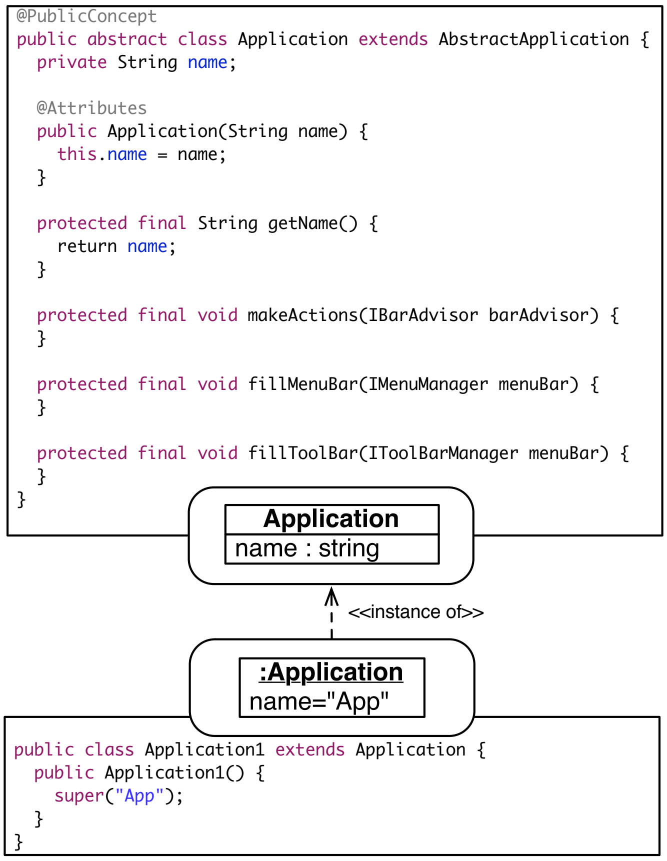
The bottom part of the figure presents an instance of the concept *application* with the attribute *name* equals to "App", together with the AA that inherits from the DORI module and implements the concept instance. Notice that the object diagram contains all the necessary information for obtaining the code in a straightforward way.

The next subsections present other DORI modules that depend on Application, namely for completing the behavior of the Composition Hook Methods.

5.4.2.2 Composite associations

In order to illustrate composite associations, consider the *menu* concept as a child of *application*. Using a Self-Pluggable Object (Section 4.5) to represent a child concept is an adequate solution.

Figure 5.9 presents a DORI module that expresses the *menu* concept. A menu was considered a non-public concept, and therefore is annotated with @Concept. Analogously to the case of Application, this module defines an attribute representing its *name*. It has a method createMenu() that creates an instance of MenuManager using the name attribute. The method addItems() is a Composition Hook



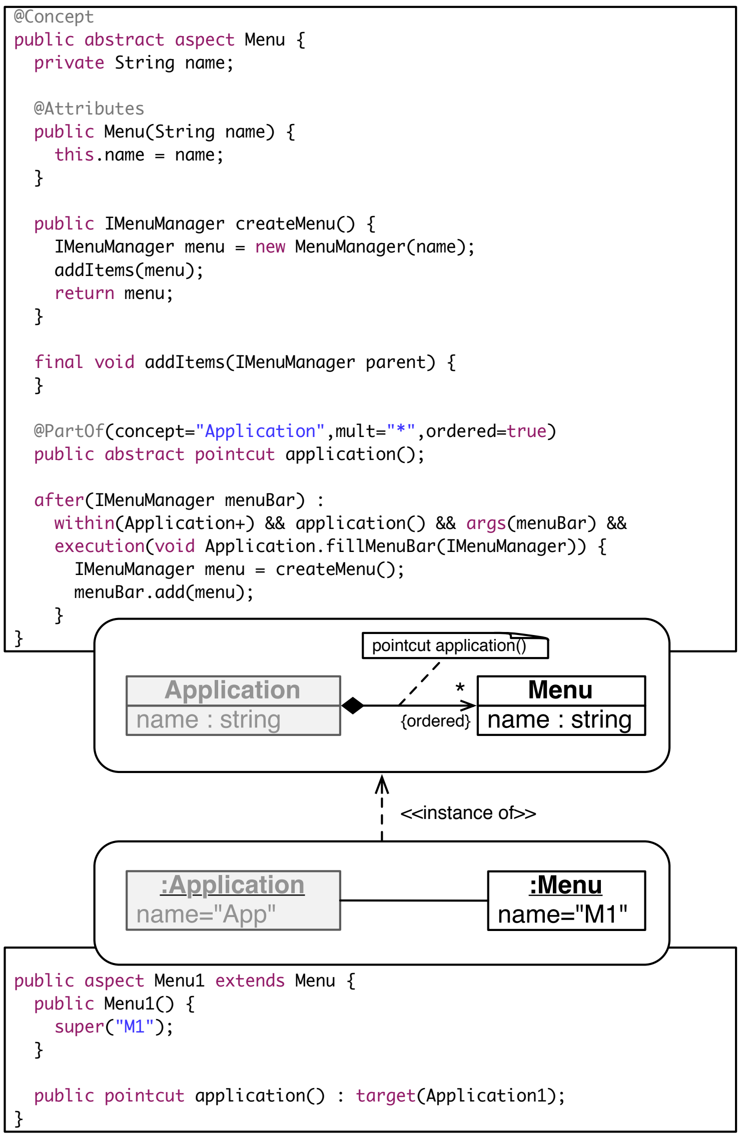
```
@PublicConcept
public abstract class Application extends AbstractApplication {
  private String name;

  @Attributes
  public Application(String name) {
    this.name = name;
  }

  protected final String getName() {
    return name;
  }

  protected final void makeActions(IBarAdvisor barAdvisor) {
  }

  protected final void fillMenuBar(IMenuManager menuBar) {
  }

  protected final void fillToolBar(IToolBarManager menuBar) {
  }
}
```

```
public class Application1 extends Application {
  public Application1() {
    super("App");
  }
}
```

Figure 5.8: Expressing concepts in DORIs.

METHOD, which analogously to the case of Application, is left empty and locked. This method will be handled by the DORI module that represents the *menu items* (to be introduced later). The parent concept of the composite association is given by the DORI module Application, implying that each application may contain several menus, which are ordered. The class diagram shows the fragment of the domain variability model that the code represents. Notice the attachment of an annotation on the composite association which indicates the pointcut that enables the composition of a *menu* in an *application*.

The bottom part of the figure shows an application model that includes a menu with name “M1” in the application that was already defined in the previous subsection. The AA that represents the menu defines the pointcut application() on the module Application1 (given in Figure 5.8). The application model contains



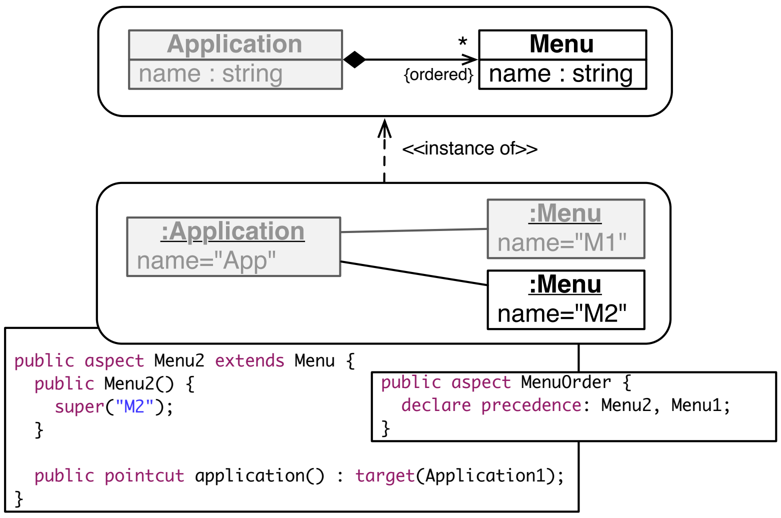
```
@Concept
public abstract aspect Menu {
  private String name;

  @Attributes
  public Menu(String name) {
    this.name = name;
  }

  public IMenuManager createMenu() {
    IMenuManager menu = new MenuManager(name);
    addItems(menu);
    return menu;
  }

  final void addItems(IMenuManager parent) {
  }

  @PartOf(concept="Application",mult="*",ordered=true)
  public abstract pointcut application();

  after(IMenuManager menuBar) :
    within(Application+) && application() && args(menuBar) &&
    execution(void Application.fillMenuBar(IMenuManager)) {
      IMenuManager menu = createMenu();
      menuBar.add(menu);
    }
}
```

```
public aspect Menu1 extends Menu {
  public Menu1() {
    super("M1");
  }

  public pointcut application() : target(Application1);
}
```

Figure 5.9: Expressing composite associations in DORIs.

all the required information for obtaining the code. The name of the pointcut application() can be obtained from the annotation on the association between the classes that represent the concepts.

An instance of a parent concept may contain several instances of child concepts, as seen in the given example. Depending on the situation, the order of the several children may be relevant. For instance, in GUI applications it is relevant which menus appear first on the menu bar. Suppose an application module Menu2 defined analogously to the module Menu1 of Figure 5.9, also having Application1 as parent. Given that the menu order is relevant, it is missing the information that determines the order of the menus. Since the order of the menus corresponds to the order of execution of the invocations of IMenuManager.add(), the menu that is plugged first is the one that appears first on the menu bar. Therefore, it is necessary to control the order of advice execution between Menu1 and Menu2. This is done using aspect precedences. Figure 5.10 illustrates this situation for the case of having menu "M1" appearing first. The precedence declaration in the module MenuOrder defines that Menu2 has the highest precedence, which in case of after advices results in being executed last. Although the object diagram does not contain explicit information about the order, it is assumed that it is possible to explicitly specify the order of the child objects of a certain parent object.



```
public aspect Menu2 extends Menu {
  public Menu2() {
    super("M2");
  }

  public pointcut application() : target(Application1);
}
```

```
public aspect MenuOrder {
  declare precedence: Menu2, Menu1;
}
```

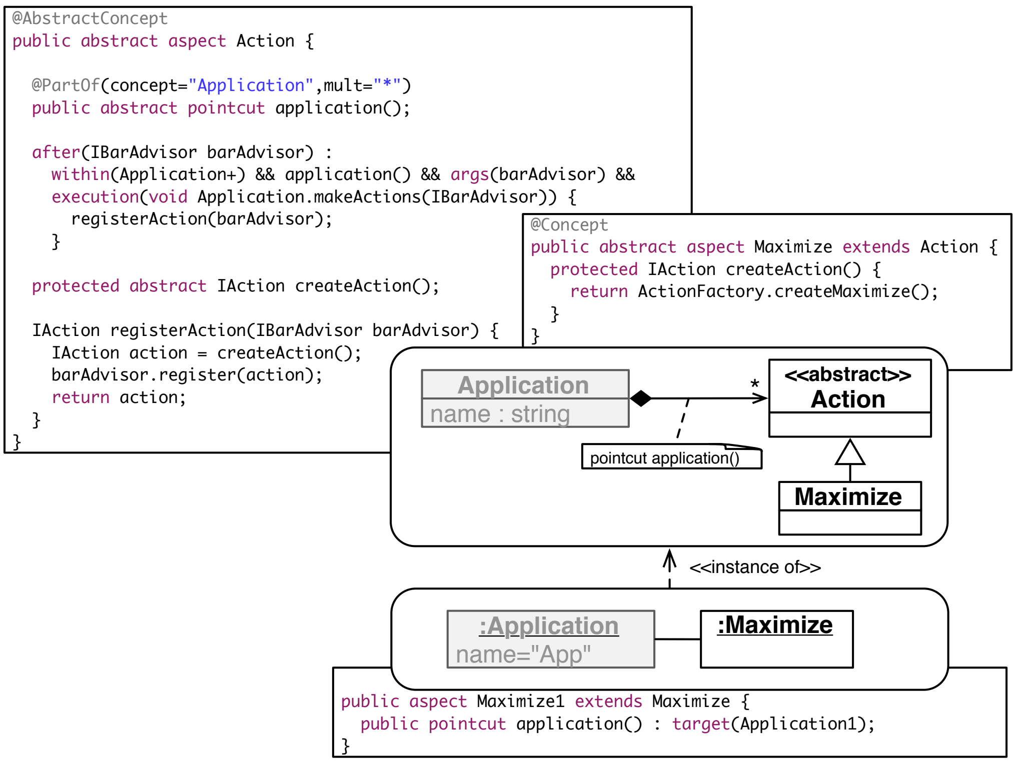
Figure 5.10: Ordering of association links.

5.4.2.3 Abstract concepts and inheritance

In order to illustrate abstract concepts and inheritance, consider the case of the abstract concept *action* and the concrete concept *maximize*. Using an Abstract Self-Pluggable Object (Section 4.7) for representing an abstract concept is an adequate solution.

Figure 5.11 presents two DORI modules — Action and Maximize. Although it might not be always the case, a DORI module expressing an abstract concept is likely to have abstract methods, which are to be defined by its extensions. The Action module defines a composite association as in the previous Menu module, but it contains an abstract method createAction(), which is a Factory Method (Gamma *et al.*, 1995) for obtaining an instance of IAction. Since every action is plugged in the application in the same way, the several DORI modules that inherit from Action only have to provide instances of the specific actions by defining createAction(). As an example of such an extension, Figure 5.11 presents the module Maximize. It implements createAction() for returning the *maximize* action using ActionFactory. Notice that the abstract pointcut application() declared in Action is still to be defined in the AAs that inherit from Maximize.

The figure shows the fragment of the domain variability model that the two



```
@AbstractConcept
public abstract aspect Action {

  @PartOf(concept="Application",mult="*")
  public abstract pointcut application();

  after(IBarAdvisor barAdvisor) :
    within(Application+) && application() && args(barAdvisor) &&
    execution(void Application.makeActions(IBarAdvisor)) {
      registerAction(barAdvisor);
    }

  protected abstract IAction createAction();

  IAction registerAction(IBarAdvisor barAdvisor) {
    IAction action = createAction();
    barAdvisor.register(action);
    return action;
  }
}
```

```
@Concept
public abstract aspect Maximize extends Action {
  protected IAction createAction() {
    return ActionFactory.createMaximize();
  }
}
```

```
public aspect Maximize1 extends Maximize {
  public pointcut application() : target(Application1);
}
```

Figure 5.11: Expressing abstract concepts and inheritance in DORIs.

DORI modules are expressing, and an example application model fragment that instantiates it. The example defines the inclusion of the *maximize* action in the *application* "App" defined previously. In the code we can see the pointcut application() matching the module Application1 (introduced in Figure 5.8).

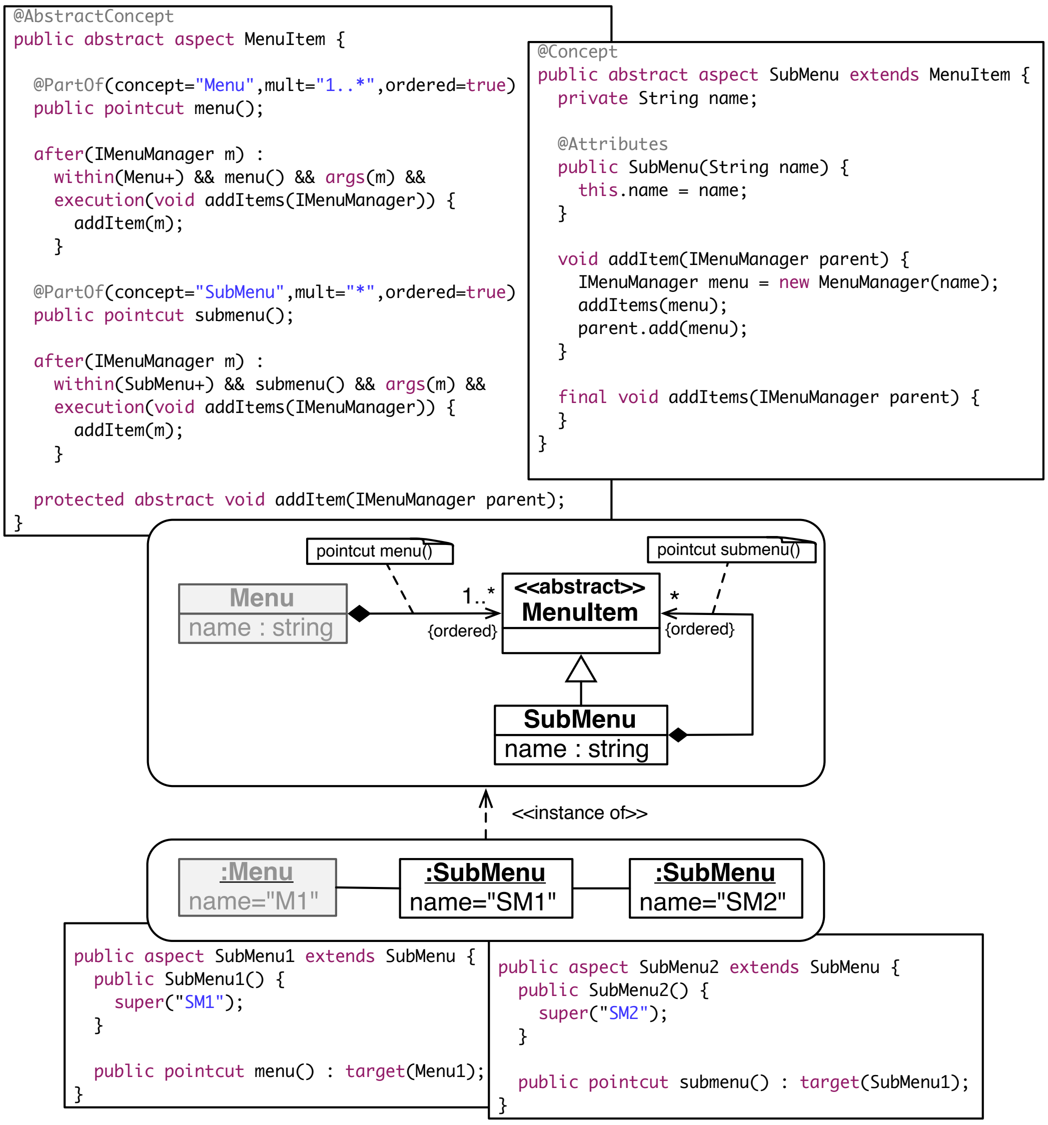
5.4.2.4 Multi-parent child concepts

In order to illustrate multi-parent child concepts, consider the concept *menu item*, which can be a child of either *menu* or *submenu*. Using a MULTI-CONTEXT SELF-PLUGGABLE OBJECT (Section 4.6) for representing a multi-parent child concept is an adequate solution.

DORI modules that express a multi-parent child concept are similar to the ones that express normal child concepts, except that they declare several parent concepts. Figure 5.12 presents a DORI module MenuItem that expresses an abstract concept with two parent declarations — one with the *menu* as parent and another with the *submenu* as parent. The mechanism for composition is the same as shown before, but the pointcuts are not abstract and they are empty (no join points are matched). The idea is to redefine the proper pointcut according to the parent concept of the *menu item*. Figure 5.12 also presents a DORI module SubMenu expressing the concept of a *submenu*, as a subaspect of MenuItem.

The application model fragment shown in the figure describes the inclusion of the submenu "SM1" in the menu "M1" (introduced in Figure 5.9), and the inclusion of a submenu "SM2" in "SM1". The figure contains also the corresponding AAs. Notice that an AA that inherits from SubMenu only defines the appropriate pointcut, i.e. either menu() or submenu(), according to the type of its parent concept instance.

It can happen that the same concept (e.g. a menu) has different classes representing it, according to the context where it is used (e.g. a class for a normal menu, and another class for a pop-up menu). In any case, the way those concepts are used in different contexts of an application is likely to be different (e.g. the objects have to be plugged into different objects). In the example shown in this subsection, neither the class nor the way the object is plugged into differs, since the concepts *menu* and *submenu* are implemented in the same class.



```
@AbstractConcept
public abstract aspect MenuItem {

  @PartOf(concept="Menu",mult="1..*",ordered=true)
  public pointcut menu();

  after(IMenuManager m) :
    within(Menu+) && menu() && args(m) &&
    execution(void addItems(IMenuManager)) {
      addItem(m);
    }

  @PartOf(concept="SubMenu",mult="*",ordered=true)
  public pointcut submenu();

  after(IMenuManager m) :
    within(SubMenu+) && submenu() && args(m) &&
    execution(void addItems(IMenuManager)) {
      addItem(m);
    }

  protected abstract void addItem(IMenuManager parent);
}
```

```
@Concept
public abstract aspect SubMenu extends MenuItem {
  private String name;

  @Attributes
  public SubMenu(String name) {
    this.name = name;
  }

  void addItem(IMenuManager parent) {
    IMenuManager menu = new MenuManager(name);
    addItems(menu);
    parent.add(menu);
  }

  final void addItems(IMenuManager parent) {
  }
}
```

```
public aspect SubMenu1 extends SubMenu {
  public SubMenu1() {
    super("SM1");
  }

  public pointcut menu() : target(Menu1);
}
```

```
public aspect SubMenu2 extends SubMenu {
  public SubMenu2() {
    super("SM2");
  }

  public pointcut submenu() : target(SubMenu1);
}
```

Figure 5.12: Expressing multi-parent child concepts in DORIs.

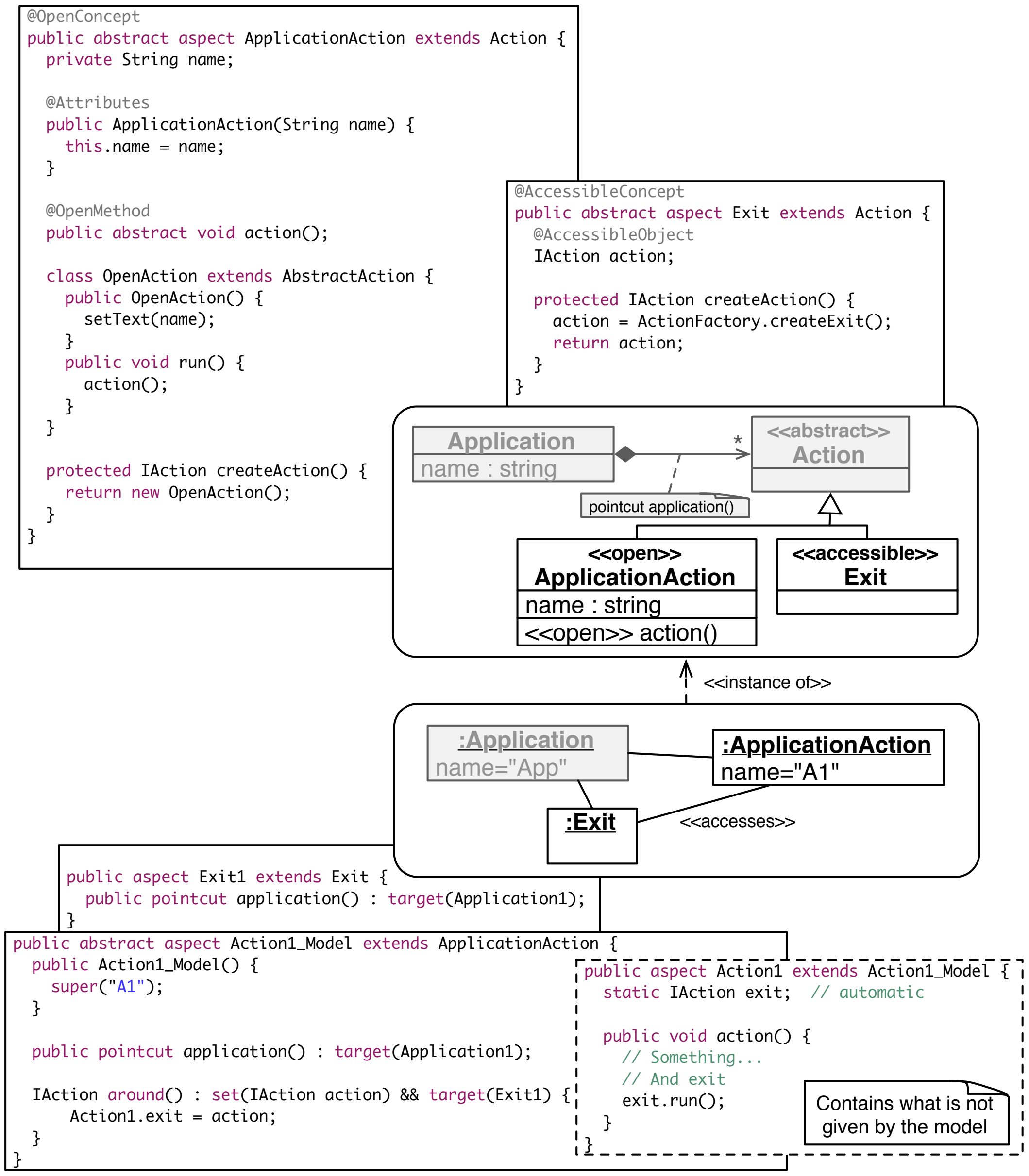
5.4.2.5 Open and accessible concepts

In order to illustrate open and accessible concepts, consider the case of the concepts *application action* and *exit*. When using an open concept, the behavior of an open method is not described in the application model. Instead, there is a dedicated module for encapsulating the method code that has to be manually defined.

Figure 5.13 presents the definition of the open concept *application action* in a DORI module that inherits from Action and defines action() as an open method. Also inheriting from Action, the figure presents the DORI module Exit representing the *exit* action as an accessible concept, defining that the accessible object is given by the attribute action. The domain variability model fragment shows the two concepts specializing the *action* concept.

As explained, instances of an open concept may access objects associated with instances of accessible concepts. In the application model we can see an instance of Exit, and an instance of ApplicationAction that defines an access to the former (association link with stereotype <<accesses>>). This association link has no corresponding relationship in the class diagram. Suppose that every instance of an open concept may define such access association links to instances of accessible concepts. Chapter 6 gives more detail on this issue.

In the code that implements the application model we can see the AA Exit1 that includes the *exit* action in the application "App" (defined in Figure 5.8). The other two AAs are associated with the inclusion of the *application action*. The module Action1_Model is an abstract aspect that implements what is expressed in the model, while it does not implement the open method. The module Action1 is the AA that completes Action1_Model, and is intended to be manually changed, namely the body of the open method action(). The last part of Action1_Model is an advice that captures the write accesses to the attribute action of Exit1, and after the object reference is changed, it sets the static variable exit of Action1 to point at the same object. Using this mechanism, the *exit* action is available to be used in the manually written code, without the need of understanding code that can be obtained from the models. The AA Action1 is able to access the exit action, as exemplified in the code.



```
@OpenConcept
public abstract aspect ApplicationAction extends Action {
  private String name;

  @Attributes
  public ApplicationAction(String name) {
    this.name = name;
  }

  @OpenMethod
  public abstract void action();

  class OpenAction extends AbstractAction {
    public OpenAction() {
      setText(name);
    }
    public void run() {
      action();
    }
  }

  protected IAction createAction() {
    return new OpenAction();
  }
}
```

```
@AccessibleConcept
public abstract aspect Exit extends Action {
  @AccessibleObject
  IAction action;

  protected IAction createAction() {
    action = ActionFactory.createExit();
    return action;
  }
}
```

```
public aspect Exit1 extends Exit {
  public pointcut application() : target(Application1);
}
```

```
public abstract aspect Action1_Model extends ApplicationAction {
  public Action1_Model() {
    super("A1");
  }

  public pointcut application() : target(Application1);

  IAction around() : set(IAction action) && target(Exit1) {
      Action1.exit = action;
  }
}
```

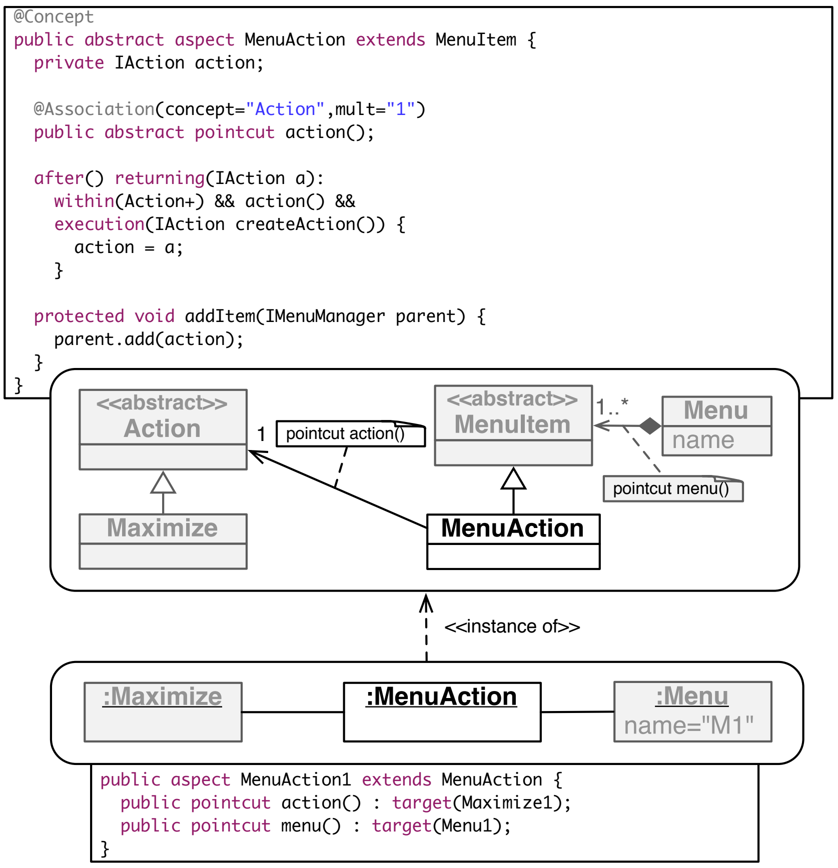
```
public aspect Action1 extends Action1_Model {
  static IAction exit;  // automatic

  public void action() {
    // Something...
    // And exit
    exit.run();
  }
}
```

Figure 5.13: Expressing open concepts in DORIs.

5.4.2.6 Directed associations

In the given domain variability model we can see two examples of directed associations: the many-to-one association between *menu action* (source) and *action* (target), and the many-to-many association between *toolbar* (source) and *action* (target). Using an Association Object (Subsection 4.9) for representing a directed association is an adequate solution. The two cases of many-to-one and many-to-many associations are discussed next.

Figure 5.14 presents the DORI module MenuAction for expressing the concept of *menu action* and the association with *action*. The many-to-one association is declared at the pointcut action(). Using method execution capturing as in previous examples, the aspect stores the object of type IAction created by create-


```
@Concept
public abstract aspect MenuAction extends MenuItem {
  private IAction action;

  @Association(concept="Action",mult="1")
  public abstract pointcut action();

  after() returning(IAction a):
    within(Action+) && action() &&
    execution(IAction createAction()) {
      action = a;
    }

  protected void addItem(IMenuManager parent) {
    parent.add(action);
  }
}
```

```
public aspect MenuAction1 extends MenuAction {
  public pointcut action() : target(Maximize1);
  public pointcut menu() : target(Menu1);
}
```

Figure 5.14: Expressing many-to-one associations in DORIs.

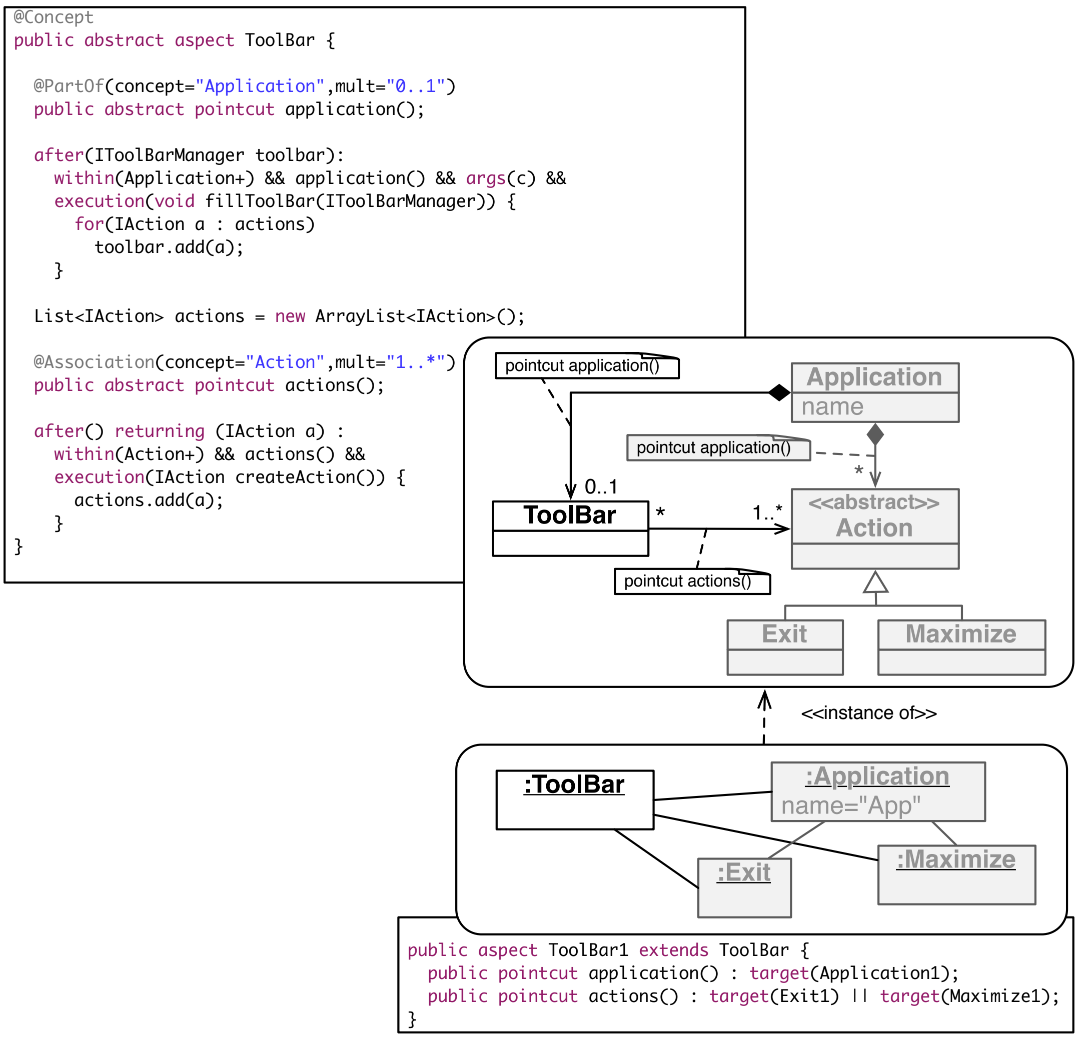
Action() of the DORI module Action. MenuAction is an extension of MenuItem, and the required implementation of addItem() plugs the stored *action* in the parent menu. The fragment of the domain variability model shows the relation of the concept MenuAction with the other concepts introduced previously.

In the bottom part of Figure 5.14 we can see an instance of the concepts and the AA that implements the inclusion of the *maximize* action in *menu* "M1" (introduced in Figure 5.9). Analogously to all the extensions of MenuItem, the pointcut menu() composes the *menu action* in the menu. The *maximize* action (introduced in Figure 5.11) becomes associated with the menu due to the definition of the pointcut action().

Many-to-many associations can be expressed in DORIs analogously to many-to-one associations. However, in case ordering is required, the solution may become problematic. Consider the case of the *toolbar*, which defines an unordered many-to-many association with the *action* concept. Despite the fact that in a general-purpose GUI framework it would make sense for this association to be ordered, let us assume that it is the framework that orders the actions in the toolbar according to some criteria.

Figure 5.15 presents a DORI module for expressing the *toolbar* concept. Regarding the capturing of *actions*, it is similar to the previous example, except that the multiplicity of the association represented by the pointcut actions() is defined to be "one or more". All the captured actions are stored in a list and used when defining the method fillToolBar(). The figure presents an application model where the *toolbar*, containing both the *exit* and *maximize* actions, is included as part of the application "App" (introduced in Figure 5.8). The toolbar is composed by defining the pointcut application() on the AA Application1, while the actions become associated by defining the pointcut actions() on the AAs Exit1 (introduced in Figure 5.13) and Maximize1 (introduced in Figure 5.11).

If this association would have to be ordered, the problem of implementing it using this type of solution is that there is no mechanism for defining the order of the several *actions* defined in the pointcut actions(). However, a many-to-many association from a source to a target concept can be converted into a composite association of a new concept in the source concept, plus a many-to-one association from the new concept to the target concept. For instance, in this example we could consider a new concept *toolbar action*, as a child of *toolbar*, that defines an association with a single *action*. Figure 5.16 illustrates this issue by presenting the original concepts and the solution based on introducing the new concept. When expressing such solution in a DORI, *toolbar actions* are children of *toolbar* and can be ordered as explained in Subsection 5.4.2.2, while the many-to-one association to action can be implemented as in the previous example.



```
@Concept
public abstract aspect ToolBar {

  @PartOf(concept="Application",mult="0..1")
  public abstract pointcut application();

  after(IToolBarManager toolbar):
    within(Application+) && application() && args(c) &&
    execution(void fillToolBar(IToolBarManager)) {
      for(IAction a : actions)
        toolbar.add(a);
    }

  List<IAction> actions = new ArrayList<IAction>();

  @Association(concept="Action",mult="1..*")
  public abstract pointcut actions();

  after() returning (IAction a) :
    within(Action+) && actions() &&
    execution(IAction createAction()) {
      actions.add(a);
    }
}
```

```
public aspect ToolBar1 extends ToolBar {
  public pointcut application() : target(Application1);
  public pointcut actions() : target(Exit1) || target(Maximize1);
}
```

Figure 5.15: Expressing many-to-many associations in DORIs.

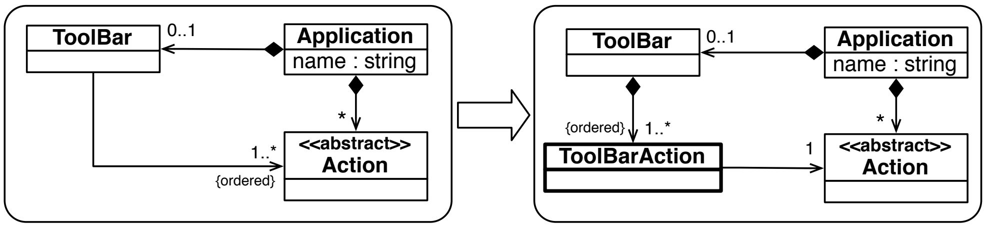


Figure 5.16: Converting a many-to-many association into a composite association plus a many-to-one association.

5.5 Discussion

This chapter explained how the several conceptual modeling constructs can be used in a DORI to express the concepts of a domain variability model. In addition to the conventional modeling constructs, special kinds of constructs were considered for dealing with the integration of manual and generated code.

The proposed mechanisms to represent the domain variability model naturally have limitations and restrictions, as for instance, the presented problem of defining ordered many-to-many associations. The way to represent concepts, their attributes and relationships, is not flexible. Each of these elements has a single and uniform mechanism for representing it. With respect to the concept instances represented in AAs, they are also represented uniformly. A concept instance is represented by a module that inherits from an SA (that represents the concept), an attribute value is represented by an argument passed to the constructor of the SA, a composite association link is represented by a pointcut that matches the parent concept instance, and a directed association link is represented by a pointcut that matches the target concept instances. Such uniformity enables the existence of a generic procedure for transforming application models into AAs, applicable to every DORI that adheres to the given representation of concepts and concept instances. By having a generic transformation, a DORI is the only artifact that one has to implement in order to have a working F-DSL. Chapter 6 addresses this issue with more detail.

Chapter 6

Automated Domain-Specific Modeling Languages

This chapter addresses the construction of DSLs based on DORIs, focusing on the specific case of *domain-specific modeling languages* (DSM Forum, 2007) (DSMLs). Section 6.1 outlines the components of a tool for building DORI-based DSLs. Section 6.2 goes into detail on a concrete solution for building DORI-based DSMLs. Section 6.3 presents ALFAMA, which is a tool prototype that realizes the proposed solution. Finally, Section 6.4 concludes the chapter by discussing the advantages of using a language workbench such as ALFAMA.

6.1 Tool Support for Building DSLs using DORIs

Chapter 5 introduced the concept of DORI. The framework usage support provided by a DORI may be regarded as an *internal DSL* (Fowler, 2008) (also known as *embedded DSLs* (van Deursen *et al.*, 2000)). An internal DSL is an extension of a base language with domain-specific abstractions using the syntactic mechanisms available in that language. Internal DSLs have the advantage of using the compiler of the base language "as is", but they lack the advantages of *external DSLs* (Fowler, 2008), which can have custom syntax, dedicated editors, and domain-specific constraints. External DSLs are typically built using a technology that suits that purpose. Table 6.1 summarizes example DSL technologies and their artifacts for defining the application models.

Technology	DSL Definition	Application Model
EMF (Eclipse Foundation, 2007d)	Meta-model	Object model
UML (OMG, 2004)	Structural Profile	Class diagram
XML (W3C, 2008a)	Schema	XML file
YACC (Johnson, 1979)	Grammar	Text file

Table 6.1: Examples of suitable technologies for developing DSLs.

Besides being an internal DSL, a DORI is beneficial for developing an external F-DSL. The abstract syntax of the F-DSL is defined by the domain variability model that is expressed in a DORI as a conceptual model. Such a conceptual model can be translated into a DSL definition using a specific technology (examples in Table 6.1). In turn, application models described using such a technology have to be transformed into DORI-based code (i.e. AAs). The examples given throughout Section 5.4 have shown how straightforward this task is, if the DORI modules are implemented as proposed. The DSL definition and transformation of application models can be automated by a tool. Figure 6.1 presents the components that are necessary to realize such a tool.

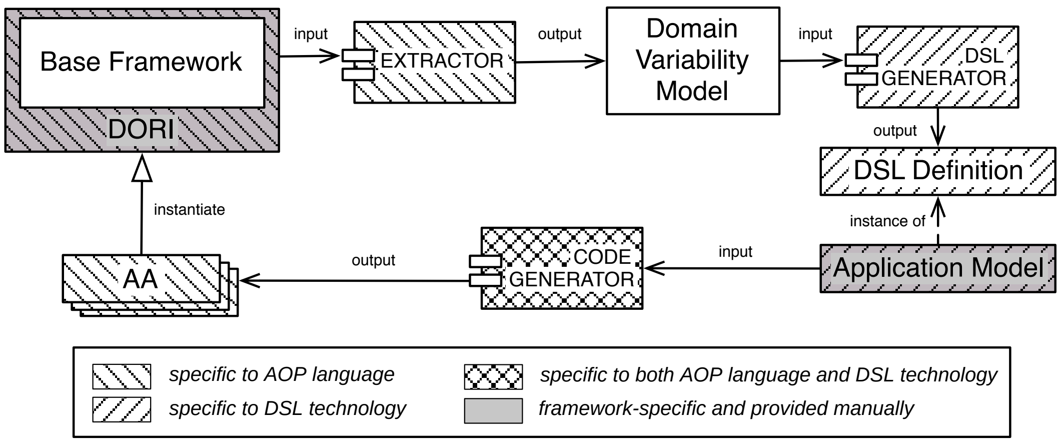


Figure 6.1: Components of a tool for building DORI-based DSLs.

The following list details the role of each component:

- *Extractor*. The role of this component is to extract the domain variability model that is expressed in the DORI modules. This model should be defined in a format that is suitable for representing conceptual models. The component is specific to the AOP language that is used to implement the DORI, while it can be used for several DORIs that are implemented in that language and that adhere to the same conventions.

- *DSL Generator.* This component takes a domain variability model as input and produces a DSL definition based on a certain technology. Therefore, this component is specific to a DSL technology, while it is independent from the AOP language, given that it deals with an implementation-independent domain variability model. In certain cases, the extractor may also fulfill the role of the DSL generator. This happens when the domain variability model is represented in a technology suitable for DSL implementation (e.g. EMF, Eclipse Foundation, 2007d). If that is the case, the domain variability model is the DSL definition itself.

- *Code Generator.* This component takes as input application models expressed in a certain DSL and produces the AAs that constitute the application code. Therefore, it is specific to both an AOP language and a DSL technology. However, its reusability is still high, given that the same code generator can be used in all situations that have in common the AOP language and the DSL technology. The code generator depends on the DSL generator, in the sense that it is built according to certain assumptions about the way the DSL generator defines the languages.

Recall that for fixed AOP and DSL technologies, all the artifacts presented in Figure 6.1 except the DORI and application models, are either generic (Extractor, DSL Generator, Code Generator) or they can be obtained automatically by a tool (domain variability model, DSL definition, AAs). By using a tool with the proposed components, the construction of an F-DSL relies solely on the DORI, while a framework-based application can be produced just by giving an application model.

The following list presents concrete scenarios using different technologies:

1. A tool composed of an extractor for DORIs written in AspectJ that outputs EMF models representing the domain variability models, and a code generator that transforms instances of EMF models into AAs written in

AspectJ. This tool is a case where the domain variability model is itself the DSL definition.

2. A tool composed of the same extractor as in (1), a DSL generator that produces XML Schemas as the grammar for application models expressed in XML files, and a code generator that transforms XML files into AAs written in AspectJ.

3. A tool composed of an extractor for DORIs written in AspectC++ (Spinczyk *et al.*, 2002), the same DSL generator as in (2), and a code generator that transforms XML files into AAs written in AspectC++.

In order to have a proof-of-concept of the proposed tool, the work in this dissertation has concentrated on scenario (1), which is detailed in the next section.

6.2 Building DSMLs using DORIs

The use of meta-model-based DSLs is promoted by the *model-driven engineering* trend. Such DSLs are often referred to as domain-specific modeling languages (DSMLs). The work in this dissertation investigated more deeply the construction of DORI-based DSMLs.

6.2.1 Overview

The Extractor component discussed previously generates a representation of a domain variability model from a DORI. Meta-modeling technologies such as EMF are an appealing means for realizing a DSML. They allow to define languages using a rich set of modeling constructs that is equivalent to conceptual modeling with respect to expressiveness. Language definitions are given in terms of meta-models. A model that conforms to a meta-model is a valid set of *instances* of the concepts described in the meta-model. A meta-model can be *extended* by another meta-model. The modeling constructs that can be used to define a meta-model are also given by a meta-model — the *meta-meta-model*.

The advantage of meta-modeling technologies with respect to the proposed tool is that a domain variability model is itself the DSL definition. The domain variability model is a conceptual model, which assumes the role of the meta-model that defines the DSML.

The Code Generator component takes as input instances of the extracted meta-models, which all should have the same basic structure. Moreover, given that open and accessible concepts are common to all DSMLs, it makes sense to factor out such commonality. In order to do so, we can have a *common meta-model* which is extended by all the meta-models that are extracted by the Extractor component. The common meta-model is an instance of the meta-meta-model. The extracted meta-models, given that they are extensions of the common meta-model, are also instances of the meta-meta-model. In turn, the application models are instances of the extracted meta-models, and are the input for the code generator.

Figure 6.2 depicts the several elements of the solution and how they are related, indicating the figures that relate to the elements. The remainder of this section will present the meta-meta-model (Figure 6.3), the common meta-model (Figure 6.4), the algorithm performed by the Extractor component (Figure 6.5), an example of an extracted meta-model that extends the common meta-model (Figure 6.6), an example application model that is an instance of the extracted meta-model (Figure 6.7), and the algorithm performed by the Code Generator component (Figure 6.8).

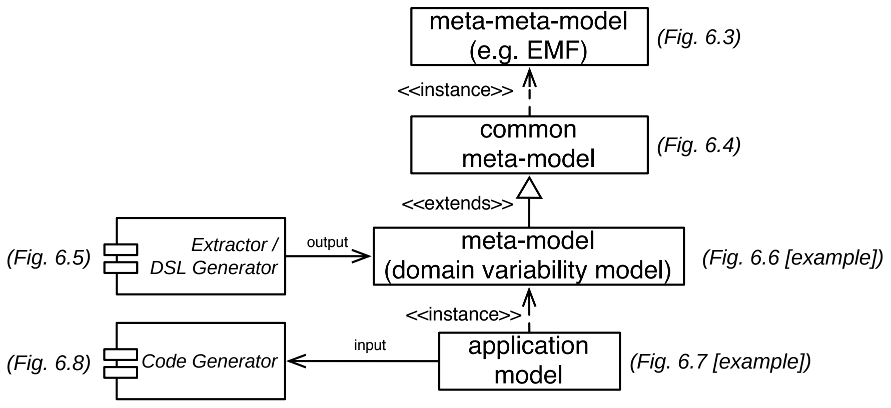


Figure 6.2: Overview of the proposed solution for building DORI-based DSMLs.

6.2.2 Meta-modeling

Figure 6.3 presents a class diagram describing a meta-model for defining meta-models. It represents a basic subset of modeling constructs that are available in meta-modeling technologies (e.g. EMF). A meta-model is composed by *classes*, which have their *name*, might be *abstract*, and might inherit from a *super* class. A class may contain several *attributes*, each one identified by a *name* and with an associated *primitive type* (e.g. *string*, *int*, etc). A class may also contain *associations* with other classes. An association has a *multiplicity*, it may be *ordered*, and it may represent a *containment* (composite association). Modeling elements such as classes and associations may contain *annotations* that store string *values*. The given modeling constructs are the ones that are supported by the proposed approach. Every DSML (meta-model) that is obtained from the Extractor / DSL Generator component conforms to the given meta-meta-model.

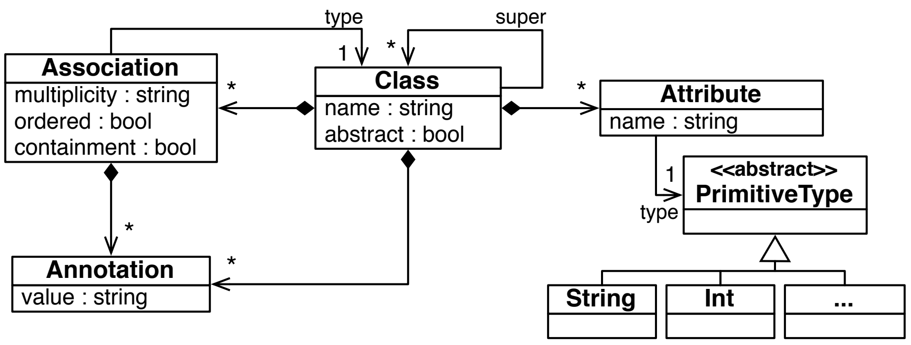


Figure 6.3: Meta-model for defining meta-models (meta-meta-model).

In Chapter 5, Figure 5.3 presented the modeling constructs that can be used to express concepts in a DORI. Such modeling constructs are closely related to the elements of Figure 6.3. The meta-model elements Class and Attribute are suitable for representing the modeling constructs Concept, OpenConcept, PublicConcept, AccessibleConcept, and AbstractConcept (implies the attribute abstract to be true). The meta-model element Association is suitable for representing the modeling constructs DirectedAssociation and CompositeAssociation (implies the attribute containment to be true).

6.2.3 Extraction of meta-models from DORIs

Figure 6.4 presents the common meta-model, which is an instance of the meta-meta-model given in Figure 6.3. The Concept is the basic unit, instantiating the element Class of the meta-meta-model. The classes are represented in UML-like class boxes, the class names in the top compartment, and the stereotype <<abstract>> denotes that the class is abstract. All elements of the extracted meta-models inherit directly or indirectly from Concept. The three special kinds of concepts are represented in their own class, namely PublicConcept, AccessibleConcept, and OpenConcept. A public concept has a filename attribute for defining the name of the file that is generated. An accessible concept is a concept that can be accessed by open concepts. An open concept may contain several Access objects to accessible concepts. A variable in the open module with name given by varname is going to point at the accessible object.

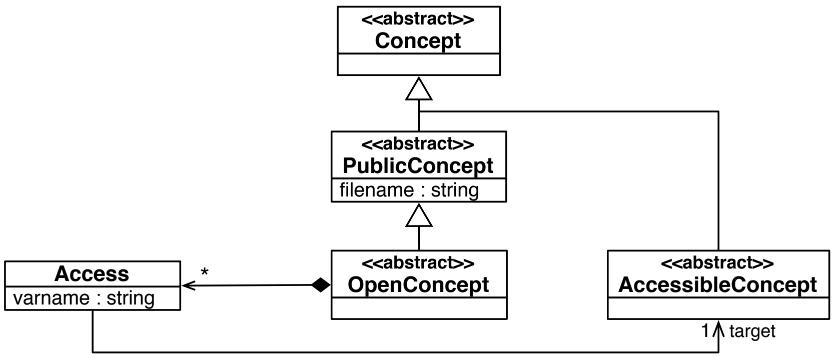


Figure 6.4: Common meta-model which is extended by all the meta-models.

Figure 6.5 presents an algorithm in pseudo-code for obtaining a meta-model from a DORI. The algorithm is rather simple, since all the necessary information is explicitly represented in the DORI modules. Each DORI module will have a corresponding class. If the DORI module does not inherit from another DORI module, its class will inherit from Concept, PublicConcept, OpenConcept, or AccessibleConcept, according to its kind. Otherwise, its class will inherit from the class corresponding to the DORI module from which it inherits. In the case of open concepts, the signatures of its open methods are attached to their class.

```
Input: DORI (set of $DORIModule$)
Output: Meta-model (set of $Class$)
foreach DORIModule dmod do
    Create $Class$ $c$ (getName(dmod));
    $c$.setAbstract(hasAbstractConceptAnnotation(dmod));
    if hasOpenConceptAnnotation(dmod) then
        $c$.addSuperClass($OpenConcept$);
        $c$.attachAnnotation(getOpenMethods(dmod));
    end
    else
        if hasAccessibleConceptAnnotation(dmod) then
            $c$.addSuperClass($AccessibleConcept$);
        end
        if hasPublicConceptAnnotation(dmod) then
            $c$.addSuperClass($PublicConcept$);
        end
    end
    foreach Attribute att in getAttributesAnnotation(dmod) do
        $c$.addAttribute($att$.getType(),$att$.getName());
    end
end
foreach Class c do
    $DORIModule$ $dmod$ = getDORIModule($c$);
    if extendsDORIModule(dmod) then
        $c$.addSuperClass(getClass(dmod.getSuper()));
    end
    if hasNoSuperClass(c) then
        $c$.addSuperClass($Concept$);
    end
    foreach PartOfAnnotation ann in dmod.getPartOfAnnotations() do
        $Class$ $parent$ = getClass($ann$.getConcept());
        Create $Association$ $ca$ ($c$, $ann$.getMultiplicity(), $ann$.isOrdered());
        $ca$.setContainment($true$);
        $ca$.addAnnotation($ann$.getPointcut());
        $parent$.addAssociation($ca$);
    end
    foreach AssociationAnnotation ann in dmod.getAssociationAnnotations()
    do
        $Class$ $target$ = getClass($ann$.getConcept());
        Create $Association$ $a$ ($target$,$ann$.getMultiplicity());
        $a$.setContainment($false$);
        $a$.addAnnotation($ann$.getPointcut());
        $c$.addAssociation($a$);
    end
end
```

Figure 6.5: Algorithm for obtaining a meta-model from a DORI.

Composite and directed associations are defined according to the information in the annotations of DORI modules.

Figure 6.6 shows the meta-model that is obtained by applying the algorithm given in Figure 6.5 to the DORI modules given throughout Section 5.4. The classes of the common meta-model are represented in gray, and the class Access (containment of OpenConcept) is not visible for reducing cluttering. Next subsection addresses the Code Generator component, which accepts instances of the extracted meta-models as input.

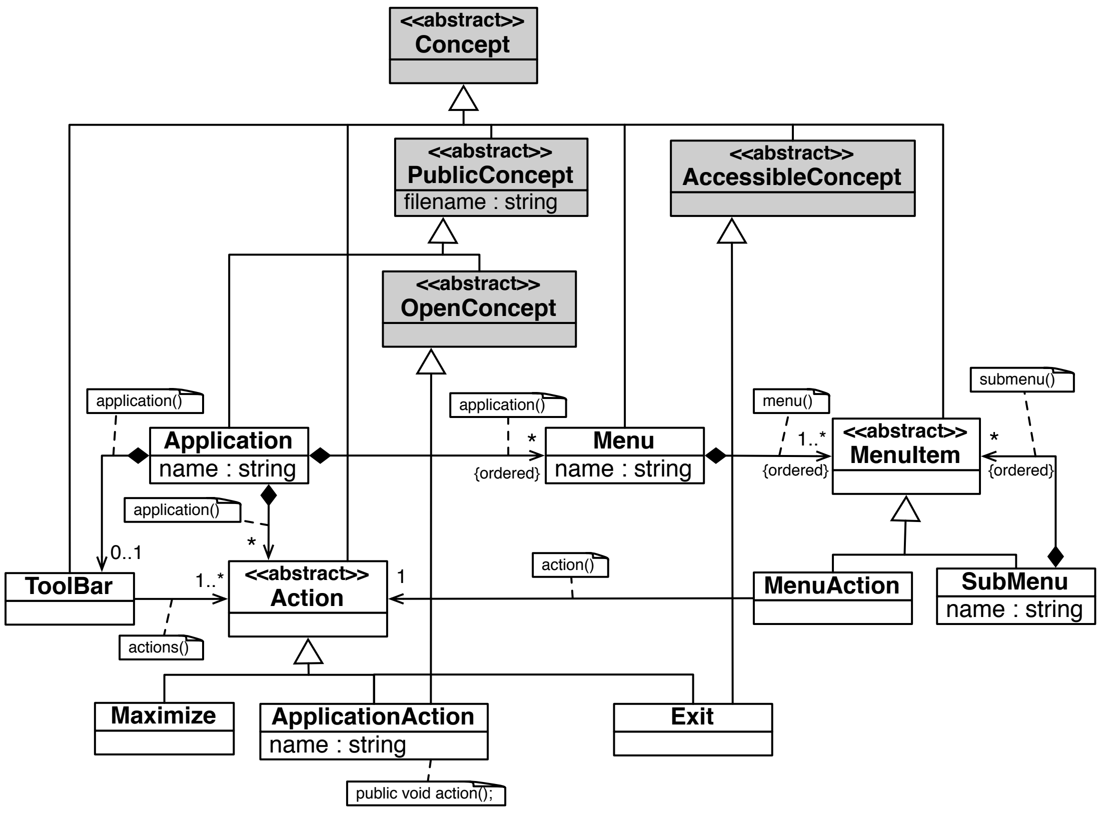


Figure 6.6: Meta-model obtained from the DORI given in Section 5.4, by applying the algorithm of Figure 6.5.

6.2.4 Generation of DORI-based code

As explained previously, the Code Generator component is generic, in the sense that it accepts instances of any meta-model extracted by the Extractor component. The Code Generator component depends on the way meta-models are extracted because it has to be aware of how certain data is represented, as for instance the pointcut names.

The code generator takes as input application models and produces application code, which is composed of AAs. Figure 6.7 presents an application model, which is an instance of the meta-model of Figure 6.6, describing the example application introduced throughout Section 5.4.

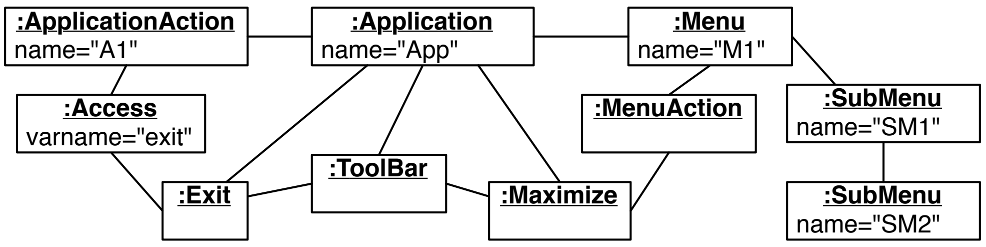


Figure 6.7: Application model (instance of the meta-model of Figure 6.6) used in the example of Section 5.4.

Figure 6.8 presents the algorithm of the generic code generator in pseudo-code. For each object in the application model there will be a corresponding AA. The name of the AAs (getUniqueName()) is given in the attribute filename in the case of a public concept or it is automatically generated otherwise. The name of the DORI module that an AA has to inherit from is equal to the name of the object's class. In the case of an open concept, an additional AA is created (the open module). Each attribute value is translated to an argument in the call to the constructor of the DORI module. Each containment association link results in a pointcut definition in the AA representing the child object. If an open module has accesses, it is augmented with a static variable with a type equal to the type of the accessible object and a name equal to the value of the attribute varname, and accordingly, the AA is augmented with an advice definition for setting the static variable to point at the accessible object. Each non-containment association link results in a pointcut definition in the AA. The examples of AAs given throughout Section 5.4 can be obtained using the given algorithm.

```
Input: Application model (set of Object)
Output: Application code (set of AA)
foreach Object obj do
    Create AA aa;
    aa.setName(getUniqueName(obj));
    aa.setSuperModule(getClass(obj).getName());
    if isOpenConcept(obj) then
        aa.setAbstract(true);
        Create AA openaa;
        openaa.setName(obj.filename);
        openaa.setSuperModule(aa);
        openaa.generateOpenMethodSkeletons();
    end
    foreach AttributeValue value in obj.getAttributeValues() do
        aa.addArgumentInSuperConstructorCall(value);
    end
    foreach AssociationLink ca in obj.getContainmentAssociationLinks() do
        Object child = ca.getTarget();
        if isAccess(child) then
            AccessibleObject ao = child.getTarget().getAccessibleObject();
            openaa.addStaticVariable(ao,child.varname);
            aa.addAdviceForStaticVariable(ao,child.varname);
        end
        else
            String pointcut = getPointcutFromAnnotation(ca);
            child.addPointcutDefinition(pointcut,aa.getName());
        end
    end
    foreach AssociationLink da in obj.getNonContainmentAssociationLinks()
    do
        Object target = da.getTarget();
        String pointcut = getPointcutFromAnnotation(da);
        aa.addPointcutDefinition(pointcut,getUniqueName(target));
    end
end
```

Figure 6.8: Algorithm for generating application code from an application model.

6.3 ALFAMA Tool

A *language workbench*, as referred by Fowler (2008), is an IDE (Integrated Development Environment) for building external DSLs. Language workbenches are especially useful when the DSLs are based on meta-modeling (DSMLs) and have custom editors for developing application models (e.g. graphical syntax). Examples of language workbenches are for instance MetaEdit+ (MetaCase, 2008) and Microsoft DSL Tools (Greenfield & Short., 2005).

As a proof-of-concept of a tool for automating the development of DORI-based DSMLs, a language workbench named ALFAMA (Automatic DSLs for using Frameworks by combining Aspect-oriented and Meta-modeling Approaches) was developed as a set of plugins for the Eclipse workbench (Eclipse Foundation, 2007b). The tool supports AspectJ (Eclipse Foundation, 2007a) as the AOP language and EMF for the DSL definitions. The following are the

The following are the three main plugins which constitute ALFAMA:

- *Meta-model extractor*. This component generates EMF models from packages containing DORI modules. The classes of the extracted meta-models inherit from the classes of a common meta-model (as the one given in Figure 6.4).

- *Generic code generator*. This component generates AAs from instances of the extracted meta-models. It uses the EMF reflection capabilities for accessing the meta-model while processing the application models.

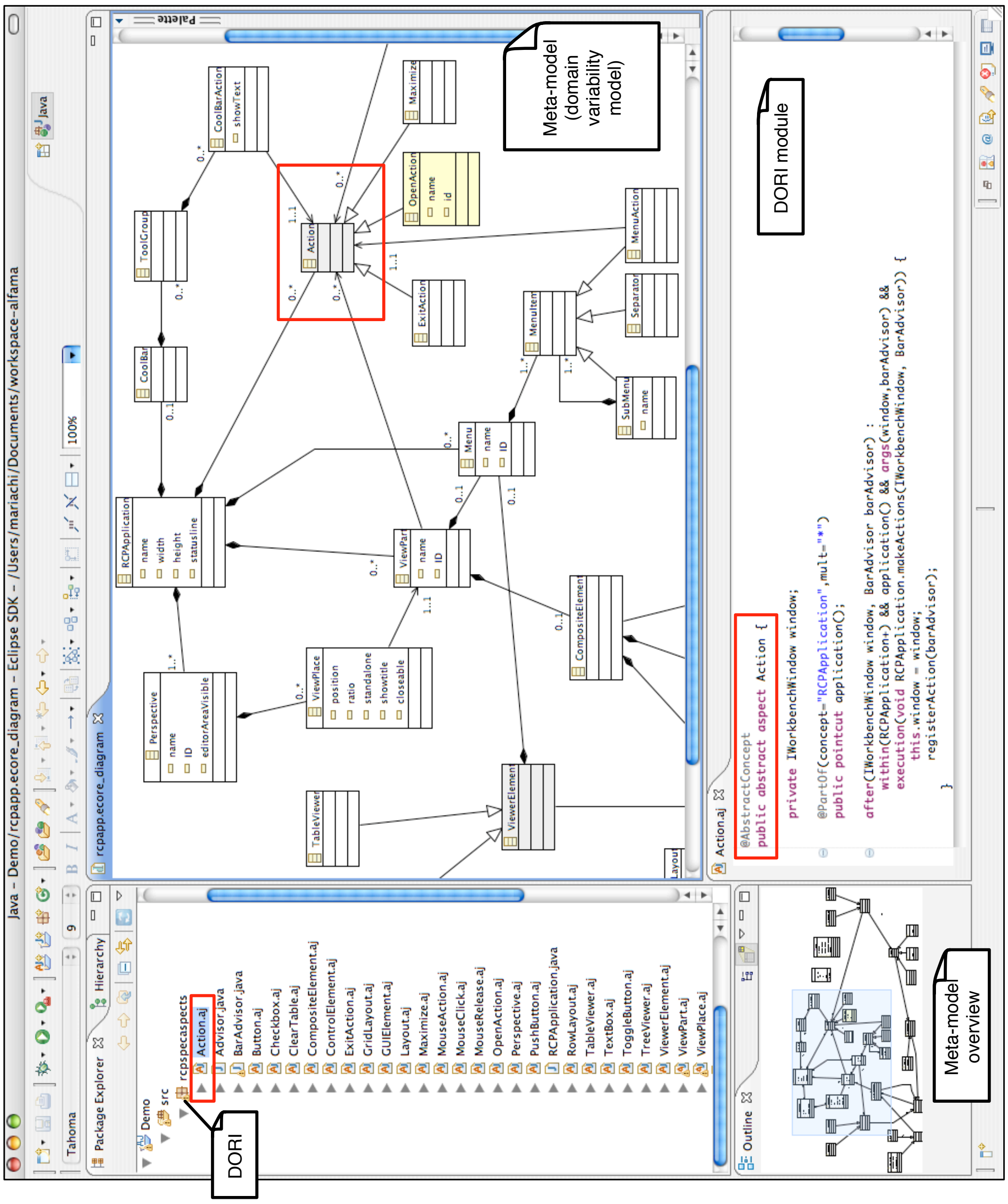
- *Integration plugin*. This component realizes the integration of the other two components with the Eclipse workbench. It includes the actions for extracting meta-models, generating code from application models, etc, plus a specific editor for application models.

Figure 6.9 shows a screenshot of the ALFAMA tool from the perspective of domain engineering. In order to develop the DSML, domain engineers have to implement the DORI for the base framework. The tool enables a one-shot extraction of the meta-model representation into an EMF model. Such a model can be visualized in a graphical way, as shown in the figure. Therefore, while implementing the DORI, the developer may constantly check how the DSML is taking shape. The graphical visualization of the DSML concepts is useful both in terms of understandability and complexity management. The DSML concepts in the graphical model can be traced to the DORI modules that represent them (one-to-one mapping).

On the left-hand side of the figure we can see the package explorer with the package rcpspecaspects containing several DORI modules for the Eclipse RCP framework. These modules are edited normally using the AspectJ editor. For instance, in the bottom part of the right-hand side we can see the DORI module Action opened in the editor. On the right-hand side of the upper part we can see a diagram representing the meta-model that is extracted from the package of DORI modules.

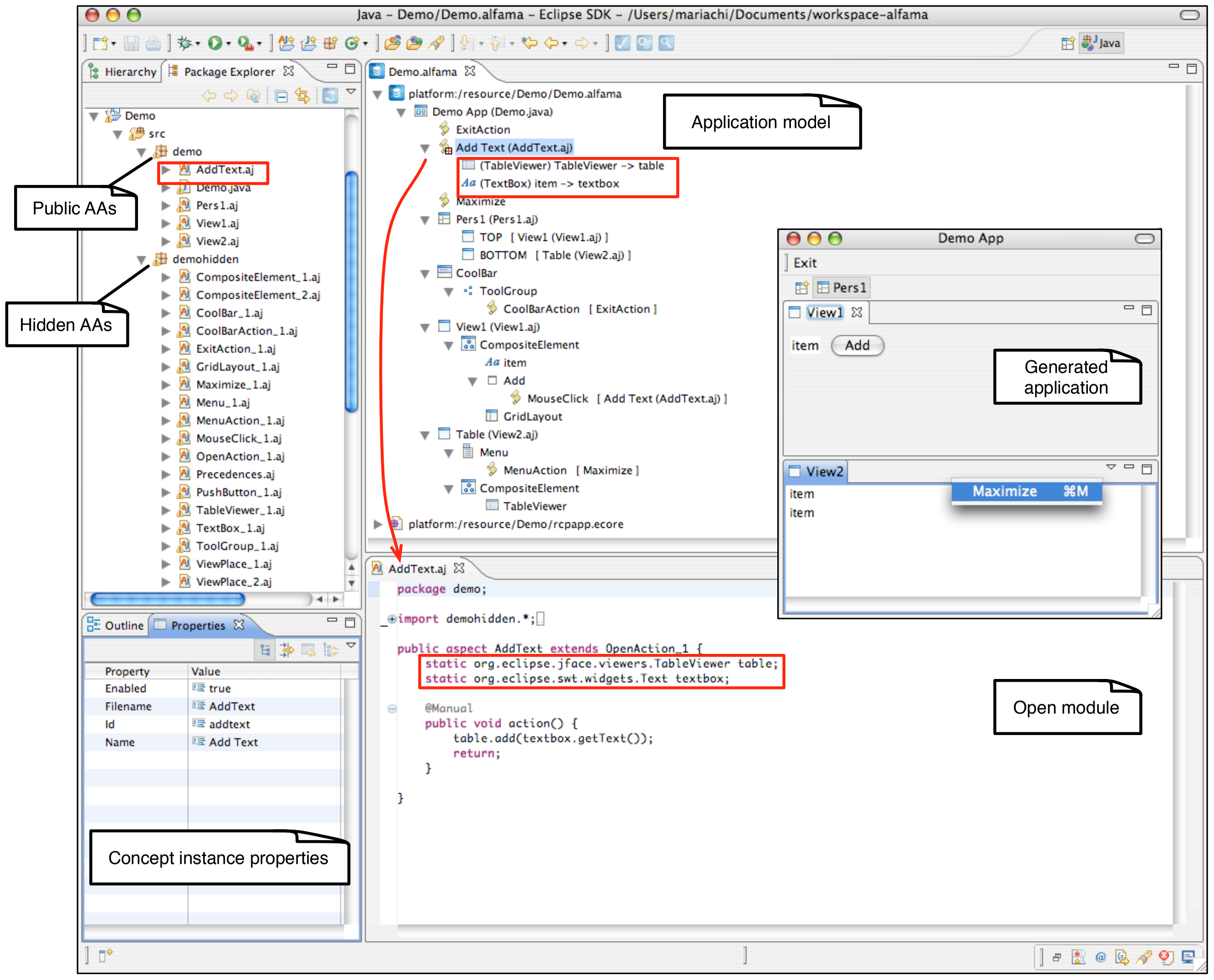
Figure 6.10 shows a screenshot of the ALFAMA tool from the perspective of application engineering using the generated DSML. The tool provides a generic tree-view editor for application models. Such an editor represents the containment of concept instances by nesting the items in the tree nodes. The editor shows icons for the concept instances, if they are specified in the DORI modules. The attribute values of the concept instances are edited in a property list. With an application model, the tool enables a one-shot generation of the AAs that implement the application. The generated code is divided in two sets: public and hidden. As the names suggest, the public set contains modules that are intended to be used externally or manipulated (open modules), whereas the hidden set contains modules that are not meant to be seen, touched, or understood by an application developer. In the case of instances of open concepts, the application developer may navigate to the open module in order to complete it manually.

On the right-hand side of the upper part of the figure we can see an application model (instance of the meta-model of Figure 6.9) describing an application based on Eclipse RCP. On the left-hand side we can see a package containing the code that was generated from the application model, divided in two packages (public and hidden). On the right-hand side of the bottom part we can see the open module AddText opened in the editor. The marks in the figure highlight how the access declarations on the application model affect the open module. The figure also shows the screenshot of the application that was generated from the model.



```
@AbstractConcept
public abstract aspect Action {

    private IWorkbenchWindow window;

    @PartOf(concept="RCPApplication",mult="*")
    public pointcut application();

    after(IWorkbenchWindow window, BarAdvisor barAdvisor) :
        within(RCPApplication+) && application() && args(window,barAdvisor) &&
        execution(void RCPApplication.makeActions(IWorkbenchWindow, BarAdvisor)) {
            this.window = window;
            registerAction(barAdvisor);
    }
```

Figure 6.9: Domain engineering with ALFAMA.



```
Demo
  src
    demo
      AddText.aj
      Demo.java
      Pers1.aj
      View1.aj
      View2.aj
    demohidden
      CompositeElement_1.aj
      CompositeElement_2.aj
      CoolBar_1.aj
      CoolBarAction_1.aj
      ExitAction_1.aj
      GridLayout_1.aj
      Maximize_1.aj
      Menu_1.aj
      MenuAction_1.aj
      MouseClick_1.aj
      OpenAction_1.aj
      Precedences.aj
      PushButton_1.aj
      TableViewer_1.aj
      TextBox_1.aj
      ToolGroup_1.aj
      ViewPlace_1.aj
      ViewPlace_2.aj
```

```
platform:/resource/Demo/Demo.alfama
  Demo App (Demo.java)
    ExitAction
    Add Text (AddText.aj)
      (TableViewer) TableViewer -> table
      (TextBox) item -> textbox
    Maximize
    Pers1 (Pers1.aj)
      TOP  [ View1 (View1.aj) ]
      BOTTOM  [ Table (View2.aj) ]
    CoolBar
      ToolGroup
        CoolBarAction  [ ExitAction ]
    View1 (View1.aj)
      CompositeElement
        item
        Add
          MouseClick  [ Add Text (AddText.aj) ]
        GridLayout
    Table (View2.aj)
      Menu
        MenuAction  [ Maximize ]
      CompositeElement
        TableViewer
platform:/resource/Demo/rcpapp.ecore
```

AddText.aj

```
package demo;

import demohidden.*;

public aspect AddText extends OpenAction_1 {
    static org.eclipse.jface.viewers.TableViewer table;
    static org.eclipse.swt.widgets.Text textbox;

    @Manual
    public void action() {
        table.add(textbox.getText());
        return;
    }
}
```

Property	Value
Enabled	true
Filename	AddText
Id	addtext
Name	Add Text

Figure 6.10: Application engineering with ALFAMA.

6.4 Discussion

This chapter addressed the automated development of F-DSLs using DORIs, focusing on the case of DSMLs. The ALFAMA tool stands as a proof-of-concept that such an approach is feasible. DSMLs engineered with a DORI are thus an alternative to conventional generative approaches where the meta-models are defined externally to the framework and a code generator for processing application models is manually developed.

When compared to conventional approaches, DORI-based DSML construction is advantageous regarding maintainability. The following scenarios illustrate such advantages:

- *Addition of new DSML concept.* In a conventional solution, the concept has to be added in the meta-model and the code generator has to be modified to support the new concept. In a DORI-based solution, a new DORI module has to be developed. Moreover, due to inheritance between DORI modules, it is possible to have design solutions where new DSML concepts can be developed without the need of understanding any other existing modules (e.g. the *actions* in the example framework).

- *Removal of DSML concept.* In a conventional solution, the concept has to be removed from the meta-model and the code generator has to be modified to remove the functionality associated with the concept. In a DORI-based solution, there is no need to understand any DORI modules, as long as one knows which is the module that represents the concept. The DORI module simply has to be excluded from the meta-model extraction.

- *DSML partitioning.* If one wants to partition the DSML, i.e. to have variations of the language that include subsets of the available concepts, a conventional solution would require to build the generator according to this requirement. Otherwise, the task would be very laborious. In a DORI-based solution, a subset of self-contained DORI modules can be used to extract a meta-model representing a subset of the DSML.

Despite the fact that ALFAMA was developed as an “independent” language workbench that supports a new way of developing DSMLs for frameworks, the approach itself is not in any way incompatible with existing language workbenches, such as MetaEdit+ or Microsoft DSL Tools. As a matter of fact, the proposed language workbench could be developed as an extension/plugin of these tools. An Extractor component would output meta-models based on the proprietary meta-model formats of the tools, while a Code Generator component would be developed against those meta-model formats using the tool-provided languages for building code generators.

The development of domain-specific concrete syntax, for instance in the form of graphical notation, is not addressed by ALFAMA. This issue can be handled independently, for instance using GMF (Graphical Modeling Framework, Eclipse Foundation, 2007e) if the meta-models are in EMF, or using the facilities of the existing language workbenches when having the meta-models in their proprietary format.

Chapter 7

Evaluation

This chapter presents two case studies that were used to investigate the practical applicability of both framework specialization aspects and DORIs. Section 7.1 revisits the JHotDraw (SourceForge, 2006) framework presented on Chapter 2, focusing on the modularity enabled by specialization aspects. Section 7.2 presents a case study on the Eclipse Rich Client Platform (RCP) framework (McAffer & Lemieux, 2005), focusing on building a DORI-based DSML using ALFAMA. Finally, Section 7.3 discusses methodological risks and limitations of the approaches.

7.1 Revisiting JHotDraw

JHotDraw was the first framework that served as a case study. This section focuses on comparing conventional JHotDraw instantiation with the use of specialization aspects.

7.1.1 Specialization aspects / DORI

JHotDraw was the first real framework where the feasibility of specialization aspects (SAs) was tested. Such an experiment was done when the concept of SAs was at an immature stage and before the concept of DORI was elaborated. However, here the results of having SAs for JHotDraw are presented according to the current maturity stage, and making use of the DORI concept.

A set of SAs was successfully developed for JHotDraw, covering all the concepts listed in Table 2.1. The SAs did not require modifications on the framework, and all of them are instances of the patterns given on Chapter 4. Based on these SAs, a DORI was developed for JHotDraw.

Domain variability model

Figure 7.1 presents a simplified domain variability model expressed by the JHotDraw DORI. From the model one can see which are the SAs, given that each concept has a corresponding DORI module, and in turn, each DORI module is implemented as an SA.

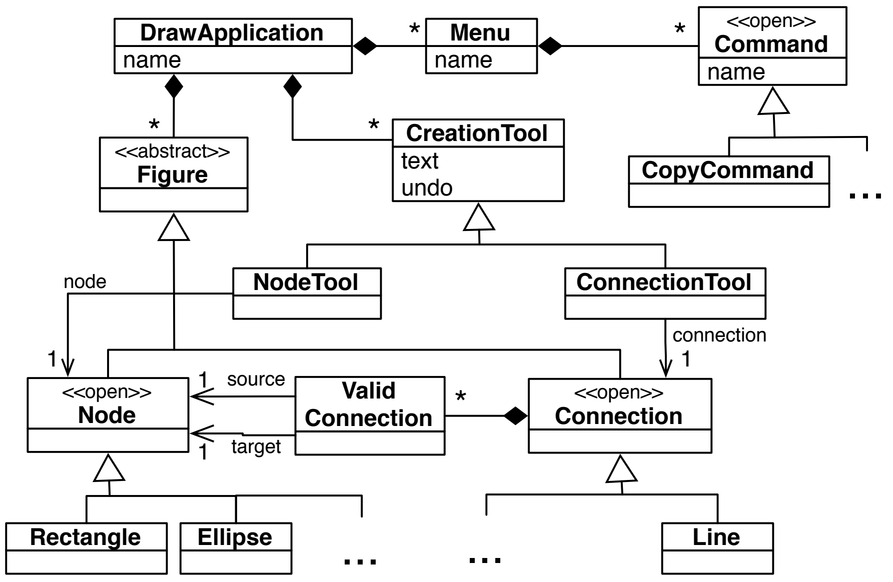


Figure 7.1: Domain variability model for JHotDraw (simplified).

When introducing JHotDraw on Chapter 2, Table 2.1 listed a set of concepts for the purpose of exemplifying the development of a DSL. For each of those concepts there was a DORI module addressing it, with the following exceptions:

- Default and application-specific node figures were merged in the NodeFigure concept, which is open for supporting application-specific figures. The default node figures are represented by concepts that inherit from NodeFigure (e.g. Rectangle).

- Default and application-specific connection figures were merged in the ConnectionFigure concept, which is open for supporting application-specific connection figures. The default connection figures are represented by concepts that inherit from ConnectionFigure (e.g. LineConnection).

- An abstract concept Figure was introduced as a generalization of NodeFigure and ConnectionFigure, given that with respect to their inclusion in the application and the use of creation tools they are handled in the same manner.

- The concept CreationTool is abstract, while it has two specializations, NodeFigure and ConnectionFigure.

- The concept Undo was incorporated as an attribute of CreationTool.

- Valid connections for default and application-specific connection figures were merged in the ValidConnection concept.

- Default and application-specific commands were merged in the Command concept, which is open for supporting application-specific commands. The default commands are represented by concepts that inherit from Command (e.g. CopyCommand).

Pattern instances

The SAs developed for JHotDraw are instances of the patterns presented in Chapter 4. Table 7.1 lists for each pattern the DORI modules that are applying it.

Pattern	**JHotDraw Concepts**
Composition Hook Method	Draw application, Menu
Self-Pluggable Object	Creation tool, Undo, Menu, Valid connection
Abstract Self-Pluggable Object	Command
Self-Pluggable Type Hierarchy	Figure and sub-concepts
Association Object	Creation tool, Valid connection

Table 7.1: Pattern instances on the specialization aspects of JHotDraw.

7.1.2 Comparison with conventional instantiation

Figure 7.2 depicts the dependencies between the application aspects (AAs) of a JHotDraw-based application. Given that the order of the *menus* and *creation tools* is relevant in a JHotDraw-based application, aspect precedences had to be used to explicitly determine the order or the several AAs that implement either menus or creation tools.

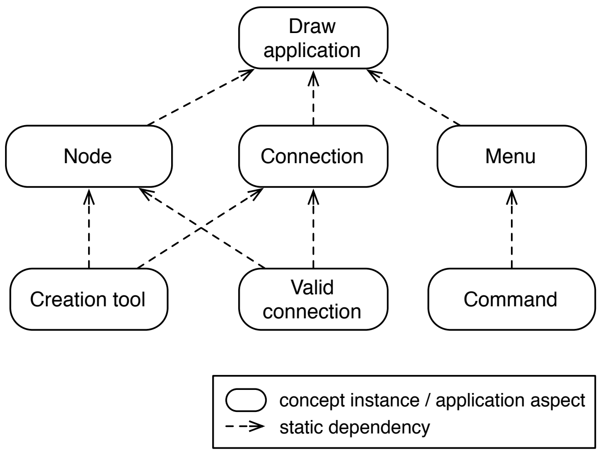


Figure 7.2: JHotDraw instantiation using specialization aspects.

In contrast to conventional JHotDraw instantiation (depicted earlier in Figure 2.4), the use of SAs has the following consequences:

- Each concept instance is implemented in its own AA, which completely implements an increment in the application.

- AAs only have static dependencies between them (i.e. the pointcut definitions). An AA does not require in any case the modification or inspection of other AAs. In the case of conventional instantiation, most of the concepts require the modification of a class originated by another concept instance.

- The logical dependencies between concept instances are reflected in the dependencies between the AAs that represent them. For instance, the *undo* (9) can be applied on a *creation tool* (8), and thus, the AA of the former has a dependency to the AA of the latter. In the case of conventional instantiation, one has to modify the main class in order to use undo on a creation tool. Therefore, the direct mapping of the logical dependency between concept instances disappears in the implementation.

- Changes in a certain concept instance are localized in the AA which represents it, so that:
  - The concept instance can be removed by not including its AA in compilation. In the case of conventional instantiation, in most cases removing a concept instance implies invasive modifications on code that is tangled with code pertaining to other concept instances (e.g. creation tools (8), undo (9)).
  - Changing something in a concept instance (e.g. moving a *command* (11-12) from a *menu* (10) to another) is done solely on the corresponding AA. In the case of conventional instantiation, most of the cases require modifying other modules that are related to other concept instances.

7.2 Case Study: Eclipse Rich Client Platform

The second case study that was carried out was based on Eclipse RCP, which is a more complex framework than JHotDraw. This section focuses on comparing conventional development of a DSML with the development of a DORI-based DSML.

7.2.1 Framework description

Figure 7.3 illustrates some of the main concepts of an *RCP application*. An RCP application is composed by several *view parts*, which are organized by *perspectives*. Each view may contain UI elements (*SWT widgets*) and viewers (e.g. *table viewer*). An RCP application may have menus, which can be *application menus* (in the menu bar), *view menus*, or *pop-up menus* in a viewer. An RCP application may have a *cool bar* (equivalent to a tool bar). An RCP application may have several *actions* whose execution is triggered by *menu actions*, *cool bar actions*, or by other UI elements such as *buttons* (associating *mouse actions*).

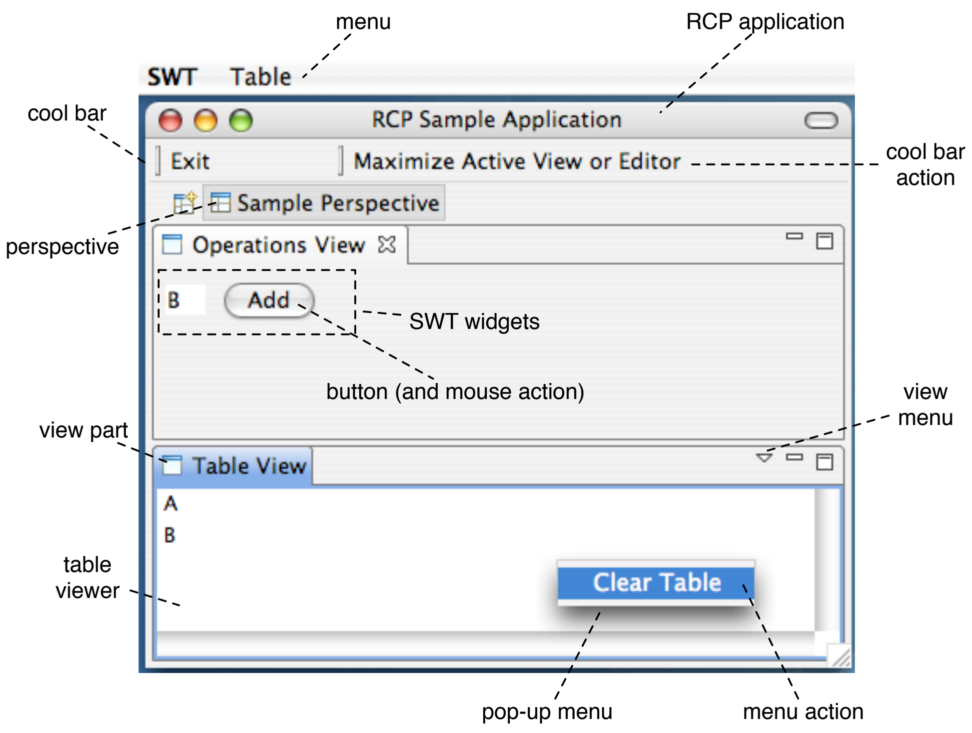


Figure 7.3: Screenshot of an application based on Eclipse RCP.

7.2.2 Conventional instantiation

Table 7.2 presents the mechanisms involved in the conventional instantiation of Eclipse RCP with respect to some of the concepts described in the previous subsection. This small set of concepts suffices for showing that developing an RCP-based application implies using all kinds of adaptation mechanisms, as well as different combinations of them.

Concept	Inheritance	Interface	Hook Method	Object Composition	Parameters
RCP application		√			
Perspective		√			
View part	√				√
Cool bar			√		
Action		√	√	√	√
Menu			√	√	√
Menu action				√	
Mouse action				√	

Table 7.2: Adaptation mechanisms for example concepts of Eclipse RCP.

7.2.3 Specialization aspects / DORI

The case study on Eclipse RCP was carried out when the concept of DORI was taking shape, and it evolved along with the approach. A DORI was successfully developed for Eclipse RCP, covering the concepts described in Subsection 7.2.1. Eclipse RCP is a relatively extensive framework, and therefore, a set of concepts had to be selected for the case study. The goal was to have a set of concepts that are frequently used by applications, and nevertheless, that all the adaptation mechanisms and modeling constructs could be covered. The ALFAMA tool was used systematically when developing the DORI, and no modifications on the Eclipse RCP framework were required.

Domain variability model

Figure 7.4 shows a fragment of the domain variability model that was expressed in the DORI. The figure shows a screenshot of the meta-model extracted by the ALFAMA tool. The gray elements represent abstract concepts, and OpenAction is an open concept. Table 7.3 presents examples where the different DORI modeling constructs are used.

Pattern instances

Table 7.4 presents examples of pattern instances in the specialization aspects of the DORI for Eclipse RCP, with respect to the concepts listed in Table 7.2.

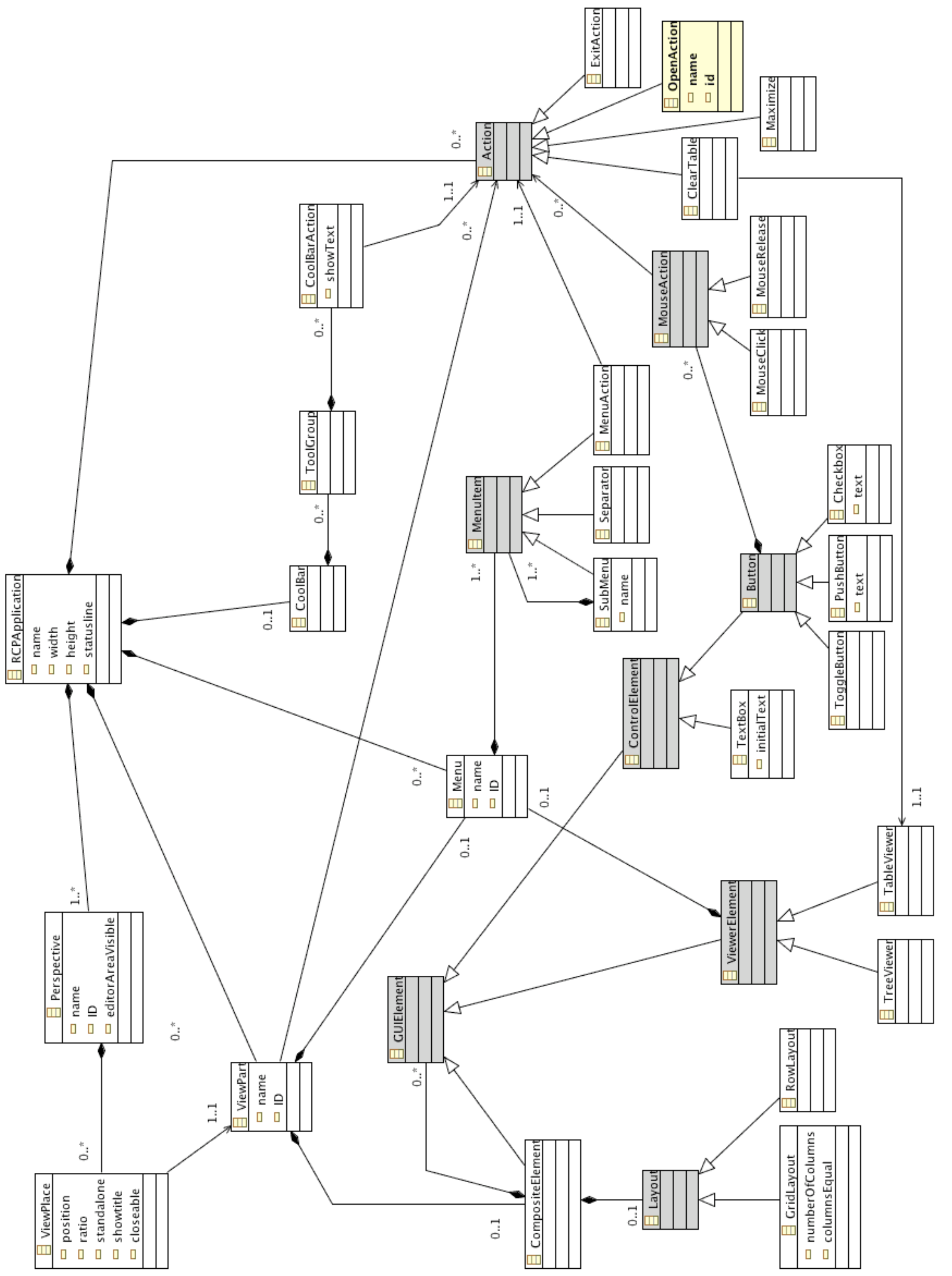


Figure 7.4: Illustrative fragment of the domain variability model of Eclipse RCP.

Modeling constructs	**Eclipse RCP Concepts**
Concepts	*All*
Composite associations	(given in *parent, child* pairs) RCP application, Perspective RCP application, View part Perspective, View place
Ordered composite associations	(given in *parent, child* pairs) RCP application, Menu Menu, Menu item
Abstract concepts	Menu item, Action, Viewer element
Multi-parent composite associations	(given in *(parent, ...), child* pairs) (RCP application, View part, Viewer element), Menu (Menu, Submenu), Menu item
Many-to-one directed associations	(given in *source, target* pairs) View place, View part Menu action, Action Clear table, Table viewer
Many-to-many directed associations	(given in *source, target* pairs) Mouse action, Action
Open concepts	Open action
Accessible concepts	Action, Viewer element

Table 7.3: Examples of modeling constructs in the DORI for Eclipse RCP.

Pattern	**RCP Concepts**
Composition Hook Method	RCP application, View part
Self-Pluggable Object	Perspective, Cool bar
Multi-Context Self-Pluggable Object	Menu
Abstract Self-Pluggable Object	Action, Mouse action
Association Object	Mouse action, Menu action

Table 7.4: Pattern instances in the specialization aspects of Eclipse RCP.

7.2.4 Comparison with conventional code generation

Conventionally, a DSML is realized by having a definition of concepts in a meta-model, and by implementing a code generator that transforms instances of the concepts into framework-based code. When using the ALFAMA tool, a DSML can be realized just by developing a DORI.

This subsection focuses on comparing the construction of DORI-based DSMLs against conventional approaches. In order to do so, a DSML was developed "conventionally" for a same framework and DSML concepts covered by an equivalent DORI-based solution. The goal was to have a solution that would allow a fairly thorough comparison with the DORI-based solution. A small fragment of the Eclipse RCP case study was selected for making the comparison.

The comparison setup went through the following steps:

1. From all the DORI modules that were developed for Eclipse RCP, a self-contained fragment of those were selected for the comparison. The criteria was to pick a simple example, which would nevertheless involve all the modeling constructs and framework adaptation mechanisms.

2. Using ALFAMA, a meta-model was extracted from the fragment of DORI modules selected in (1). Given that the meta-model is represented in EMF, Java classes implementing that meta-model can be generated. The meta-model can be accessed, manipulated, etc, using those classes.

3. Using Java, a conventional generator was implemented against the classes generated from the extracted meta-model obtained in (2). The generator was implemented in the most straightforward way, using plain Java and with no resort to code templates.

7.2.4.1 Meta-model fragment

Figure 7.5 shows the chosen meta-model fragment extracted by ALFAMA. For the concepts in the meta-model, Table 7.5 presents which modeling constructs they are using and which adaptation mechanisms are required when transforming them into framework-based code that uses the conventional reuse interface.

7.2.4.2 Conventional generator in Java

The code generator that was developed accepts as input a set of objects that is a valid instance of the meta-model given in Figure 7.5. The generator implementation is naïve, in the sense that no variability mechanisms within the generator were used. Such variability mechanisms, for instance using the Visitor pattern (Gamma *et al.*, 1995), would enable the generator to evolve more easily at predefined variation points. However, by including variability mechanisms, the generator itself would also become some sort of framework that would also have

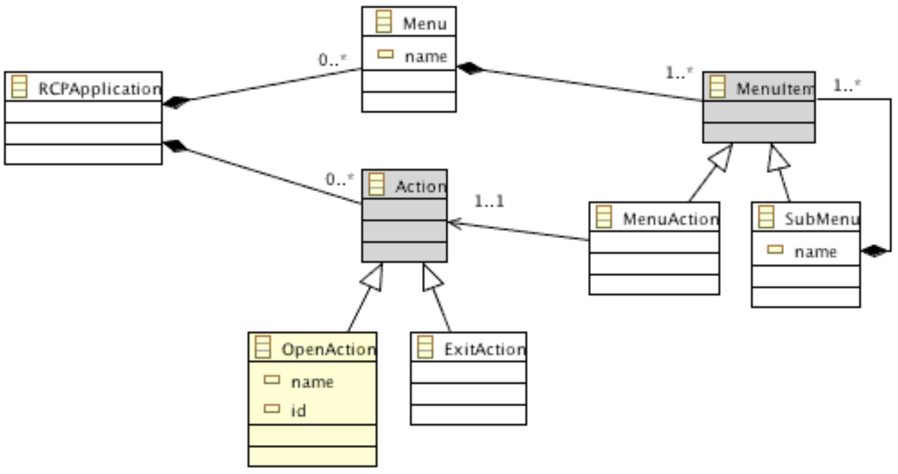


Figure 7.5: Meta-model fragment for comparison.

	Inheritance	**Interface**	**Hook Method**	**Object Composition**	**Parameters**
Independent concept		RCP app.			
Composite concept			Action, Menu	Menu item	Menu, Sub menu
Abstract concept	Action			Menu item	
Multi-parent concept				Menu item	
Directed association				Menu action	
Open concept	Open action		Open action		Open action

Table 7.5: Modeling constructs and adaptation mechanisms for the concepts of Figure 7.5.

to be learned, so that support for new concepts could be added. Moreover, the generator implementation would be more complex.

The generator was implemented against the same meta-model that was extracted by ALFAMA (Figure 7.5). Instances of the meta-model (application models) can be accessed through Java classes that implement the meta-model. Using such classes is intuitive. For each meta-model class there is a corresponding Java class. Such class has methods for accessing attributes, contained objects, and directed associations. As an example of accessing an attribute, Menu.getName()

would return the name attribute value. As an example of accessing contained objects, Menu.getMenuitem() returns references to the contained MenuItem objects. Finally, as an example of accessing a directed association, MenuAction.getAction() returns a reference to the associated Action object.

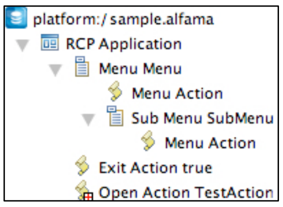
Figure 7.6 presents (a) a sample application model using the concepts of Figure 7.5 (visualized in ALFAMA), (b) the reuse interface fragment that is relevant for the concepts of the application model, and (c) application code that implements the application model using the reuse interface fragment. The application code is what the developed generator outputs for the application model. The remainder of this subsection presents the generator implementation and compares it with the DORI. Fragments of the conventional generator are introduced throughout the several subjects of comparison. The DORI modules for the given fragment of concepts were already presented throughout Chapter 5.

7.2.4.3 Modularity

In order to compare the modularity of the conventional generator and the DORI, the generation of the main application class is used as an example. Figure 7.7 presents the part of the generator that handles this issue. This generator part is actually the main generator class.

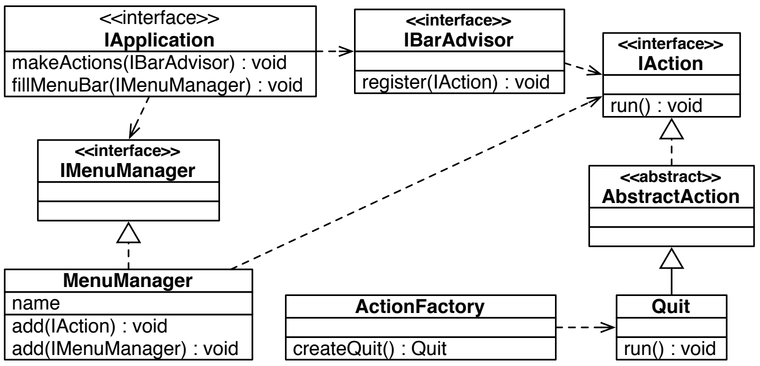
The generator can be instantiated with an RCPApplication object and the name of the main application class to be generated. The method generate() returns a list of strings, each one containing one generated application module. The hash table actionVarsTable stores the variable names that are assigned to the several actions. This is necessary because the variables referencing actions have to be shared among the hook methods makeActions(..) and fillMenuBar(..). The methods generateMakeActions() and generateFillMenuBar() handle these two hook methods, and are introduced ahead.

The generator parts for handling the actions and the menus necessarily depend on the generator part that handles the main class. This is due to the fact that both actions and menus entail a method to be implemented in the main class. Moreover, the generator part that handles the menu actions necessarily depends



```
platform:/sample.alfama
  RCP Application
    Menu Menu
      Menu Action
      Sub Menu SubMenu
        Menu Action
    Exit Action true
    Open Action TestAction
```

(a) Application model



(b) Reuse interface fragment

```java
public class Sample implements IApplication {
    private static IAction action0;
    private static IAction action1;

    public void makeActions(IBarAdvisor b) {
        action1 = new TestAction();
        b.register(action1);
        action0 = ActionFactory.createQuit();
        b.register(action0);
    }

    public void fillMenuBar(IMenuManager m) {
        MenuManager menu0 = new MenuManager("Menu");
        m.add(menu0);
        menu0.add(action0);
        MenuManager submenu0 = new MenuManager("SubMenu");
        menu0.add(submenu0);
        submenu0.add(action1);
    }
}

public class TestAction extends AbstractAction {
    public void run() {
        // TODO
    }
}
```

(c) Application code

Figure 7.6: Conventional code generation for the fragment of concepts.

```java
public class ConventionalGenerator {
   private RCPApplication app;
   private String _class;
   private Hashtable<Action, String> actionVarsTable;
   private StringBuilder appModule;
   private List<StringBuilder> outputModules;

   public ConventionalGenerator(RCPApplication app, String outputClass) {
      this.app = app;
      _class = outputClass;
      actionVarsTable = new Hashtable<Action, String>();
      appModule = new StringBuilder();
      outputModules = new ArrayList<StringBuilder>();
   }

   public List<StringBuilder> generate() {
      appModule.append("public class " + _class + " implements IApplication {\n");
      generateMakeActions();
      generateFillMenuBar();
      appModule.append("}\n");
      outputModules.add(appModule);
      return outputModules;
   }
   /* ... */
}
```

Figure 7.7: Generator part that handles the main class of the application.

on the generator part that handles the actions. By using such a generator structure it becomes difficult to modularize the generator according to meta-model concepts. If one wants to remove the support for a certain meta-model concept in the generator, one has to understand more than one generator part. Moreover, the interdependency between the generator parts makes the evolution of the generator hard.

Conventional code generators cannot be implemented in cohesive and composable modules. This implies that adding increments to the generator involves modifications in existing generator modules. When using a DORI, each meta-model concept is defined and supported by a DORI module. Such modules can be composed to form different variants of the DSML. One can make combinations of modules and obtain different DSMLs without needing to understand any internals of these modules. Although DORI modules also have dependencies between them, it is possible to evolve the DSML by developing modules without requiring changes in the existing modules. For instance, in the code of Figure 7.7 we can see several statements that are not directly related to having an empty application, such as actionVarsTable and the calls to the methods for handling the actions and menus. Moreover, although there is a separate method for handling actions and another for handling menus, the code of those methods is tied to the attributes of the ConventionalGenerator. In the solution based on DORI modules, the modules for handling actions and menus can be developed independently from the module that handles the application concept.

7.2.4.4 Extensibility

In order to compare the extensibility of the conventional generator and the DORI, the generation of code pertaining to the actions is used as an example. Figure 7.8 presents the generator method that handles this issue. This method obtains all the Action objects contained in the RCPApplication root object, assigns a name for each of the actions, stores those names in actionVarsTable, and includes an attribute for each action in the generated class (lines 2-7). Moreover, it handles the implementation of the hook method makeActions(..) in the generated class (lines 9-20), where each action attribute is initialized. Given that an Action object is either of type ExitAction or OpenAction, and these have to be handled differently, the if-else block (13-16) addresses this issue. Finally, the action instance is composed in the IBarAdvisor object using register(..) (line 18). As shown previously, generateMakeActions() method is called by the main generator method generate().

Extending the generator for supporting new action types requires at least to modify the if-else block (lines 13-16). One would have to understand the contents of the method, which is already part of a larger unit (the ConventionalGenerator class), in order to make the correct change to support a new action type. A possible way of avoiding this would be to have a solution based on the Visitor pattern, as already mentioned. However, the hot spot to adapt the generator would have to be learned. The generator would also become like a framework and it could not be used and reasoned as a regular program.

With the solution based on the DORI, new DORI modules that extend the Action module can be developed to support new actions. For instance, Figure 7.9 presents a DORI module representing a new *maximize* action. The given DORI module is the only thing that one has to define to extend the DSML with the

```
private void generateMakeActions() {
    int varid = 0;
    for(Action a : app.getAction()) {
        String varName = "action" + varid++;
        actionVarsTable.put(a, varName);
        appModule.append("private static IAction " + varName + ";\n");
    }
    appModule.append("\n");
    appModule.append("public void makeActions(IBarAdvisor b) {\n");
    for(Action a : actionVarsTable.keySet()) {
        String actionInit = "";
        String var = actionVarsTable.get(a);
        if(a instanceof ExitAction)
            actionInit = "ActionFactory.createQuit()";
        else if(a instanceof OpenAction)
            actionInit = generateOpenAction(a);
        appModule.append(var + " = " + actionInit + ";\n");
        appModule.append("b.register(" + var + ");\n");
    }
    appModule.append("}\n\n");
}
```

Figure 7.8: Generator part that handles the actions.

```
@Concept
public abstract aspect Maximize extends Action {
    public IAction createAction() {
        return ActionFactory.createMaximize();
    }
}
```

Figure 7.9: Example DORI module for a new maximize action.

new action. Just by developing a simple extension of the DORI module Action, both the concept and its support in the DSML become automatically available by redoing the meta-model extraction in ALFAMA. Such extensions not only can be done incrementally without modifying other modules, but also they do not require to understand internals of any module. Given that the extensions can be developed without inspecting Action, even a developer that does not understand anything about the DORI can augment it at these points just by knowing how to develop an extension of Action.

7.2.4.5 Accidental complexity

A code generator is a program that generates another program. In non-trivial cases, this "indirection" is a source of complexity that may cause a burden for those who have to implement and maintain the generators. Much of that complexity is accidental, i.e. not directly related with the mapping complexity, but instead with implementation details and alignment between concepts, generated code, and the actual framework reuse interface.

In order to illustrate such complexities, consider the case of the menus and menu items. Figure 7.10 shows yet another generator part for handling the menus. The method generates the code that implements the hook method fillMenuBar(..) (lines 2-14). Each Menu object contained in the RCPApplication root object is handled by generating code that instantiates a MenuManager object representing the menu, and plugs it into the IMenuManager object that is a parameter of the hook method (lines 5-8). Given that each menu may contain several menu items, each MenuItem object contained in the menu is handled by handleMenuItem(..), which is detailed later in Figure 7.11.

```
 1 private void generateFillMenuBar() {
 2    appModule.append("public void fillMenuBar(IMenuManager m) {\n");
 3    int varid = 0;
 4    for(Menu menu : app.getMenu()) {
 5       String menuVarName = "menu" + varid++;
 6       appModule.append("MenuManager " + menuVarName +
 7                        " = new MenuManager(\"" + menu.getName() + "\");\n");
 8       appModule.append("m.add(" + menuVarName + ");\n");
 9
10       int submenuVarId = 0;
11       for(MenuItem item : menu.getMenuitem())
12          submenuVarId = handleMenuItem(menuVarName, item, submenuVarId);
13    }
14    appModule.append("}\n");
15 }
```

Figure 7.10: Generator part that handles the menus.

The consistency between the code that the generator outputs and the framework reuse interface can easily be broken. A code generator produces text, which is code that instantiates the framework, such as in line 2 where the hook method signature is generated. This code is not checked against compilation until the generator is tested with sample inputs. This brings consistency problems, since a change in the framework may introduce unnoticeable errors in the code that is produced by a not up-to-date generator. If fillMenuBar(..) changes its signature, the generator does not manifest its inconsistency with respect to the framework.

The inconsistency would only be noticed when generating code from an application model that involves the menus. More concretely, the error would be noticed during the compilation of the generated code.

In a large-scale setting, the fragility of a code generator can be problematic, given that it is up to the developer to "manually" ensure the consistency of the code that the generator is programmed to produce, which potentially depends on a large set of framework classes. In contrast, if DORI modules define advices that are acting over inexistent elements, one gets compile-time warnings that inform that those modules are broken. Nevertheless, compilation errors also occur if the body of an advice is using inexistent framework elements.

Figure 7.11 shows the generator part that handles the menu items. In the case of a menu action (lines 3-7), the variable name of the referenced action is obtained, and the code that plugs the action in the menu is generated. In the case of a submenu (lines 8-16), the code for plugging a MenuManager object in the top menu (menuVarName) is generated (lines 11-13). Then, handleMenuItem(..) has to be called recursively to handle the menu items of the submenu. It is necessary to keep control of the variable names throughout the generation of the hook method fillMenuBar(..). Notice how the management of variables of both menus and submenus affects understandability. Moreover, the collaboration between the actions and the menus, i.e. the possibility to plug actions in the menus, had to be anticipated by having the attribute actionVarsTable in ConventionalGenerator.

When observing the code in both Figure 7.10 and Figure 7.11, we can notice the impact that the management of variable names has in the comprehension of the generator implementation. When using DORI modules there is no need to cope with such an additional complexity of managing variable names.

7.2.4.6 Integration of manual code

Finally, with respect to the integration of generated code with manual code, Figure 7.12 shows the generator part that handles the open actions. The method generateOpenAction(..) introduces a new module in the generation (line 10). Such a module is an extension of AbstractAction (lines 4-9), and the skeleton of the required run() method is included. This generated module is intended to be completed manually by the application developer.

As discussed earlier in this dissertation, the open action may need to access objects that are instantiated within the generated code, which is not meant to be inspected by an application developer. In these cases, there is no straightforward solution. One could have an additional “global” class that is generated, which keeps references to the objects instantiated within the generated code. Such objects could be accessed for instance by invoking a method with some *id* for the object. This kind of solutions would require all the generator parts that handle objects that can be accessed by manual code to generate also code for the global class. The proposed mechanism in DORIs for accessing objects that are

```java
private int handleMenuItem(String menuVarName, MenuItem item, int varid) {
   int localVarid = varid;
   if(item instanceof MenuAction) {
      MenuAction ma = (MenuAction) item;
      appModule.append(menuVarName + ".add(" +
                       actionVarsTable.get(ma.getAction()) + ");\n");
   }
   else if(item instanceof SubMenu) {
      SubMenu submenu = (SubMenu) item;
      String submenuVarName = "submenu" + localVarid++;
      appModule.append("MenuManager " + submenuVarName +
                       " = new MenuManager(\"" + submenu.getName() + "\");\n");
      appModule.append(menuVarName + ".add(" + submenuVarName + ");\n");
      for(MenuItem i : submenu.getMenuitem())
         localVarid = handleMenuItem(submenuVarName, i, localVarid);
   }
   return localVarid;
}
```

Figure 7.11: Generator part that handles the menu items.

```java
private String generateOpenAction(Action a) {
   String actionClass = "new " + ((OpenAction) a).getName() + "()";
   StringBuilder actionModule = new StringBuilder();
   actionModule.append("public class " + ((OpenAction) a).getName() +
                       " extends AbstractAction {\n");
   actionModule.append("public void run() {\n");
   actionModule.append("// TODO\n");
   actionModule.append("}\n");
   actionModule.append("}\n");
   outputModules.add(actionModule);
   return actionClass;
}
```

Figure 7.12: Generator part that handles open actions.

instantiated within the generated code does not require additional work besides annotating the accessible objects. Moreover, the access is declared at the modeling level, and the variables referencing the objects that one wants to access are generated into the open modules. Such variables can be used directly by application developers who do not need to know about any other details.

7.2.4.7 Size

The work in this dissertation proposed DORIs to be implemented in Java/AspectJ. Programming in AspectJ is effectively programming in Java plus aspect primitives. Table 7.6 presents the number of lines of code (LOC) that the implementation of both the conventional generator and the DORI required. The LOC are divided according to the meta-model concepts of Figure 7.5, and the Java and AspectJ LOC are discriminated for the DORI.

Given the total LOC of both cases, 87 lines for the conventional generator and 124 lines for the DORI, one could think that the DORI requires more effort given that it has approximately 42% more LOC than the conventional generator. However, the given DORI modules have approximately 40 LOC of trivial code in constructor methods, attributes, and empty hook methods. The only part of the conventional generator that can be considered trivial is the one that is given in Figure 7.12. The 124 LOC of the DORI minus these 40 LOC would roughly have the same LOC as the conventional generator. The fact that the DORI is separated in several modules, while the conventional generator is in a single class, also implies more LOC in the DORI given the extra variables and module

Concept	**Conventional (Java)**	**DORI (Java + AspectJ)**
RCP application	22	6 + 0
Action	21	10 + 6
Exit action		8 + 0
Menu	14	18 + 7
Menu item	17	5 + 13
Submenu		15 + 0
Menu action		9 + 6
Open action	12	22 + 0
total	87	124 (92 + 32)

Table 7.6: LOC for conventional generator and DORI.

headers. Finally, the DORI also embodies the definition of concepts, which in the conventional solution has to be maintained separately in the meta-model.

7.3 Discussion

The role of the case studies in this work was twofold. While investigating how to implement specialization aspects and elaborating the possible modeling constructs in DORIs, the two frameworks were used as a source to mine hot spots and test solutions. By the time when the proposed approach began to stabilize, the frameworks were used to validate the approach and a more detailed analysis was performed. The remainder of this section presents some criticism to the evaluation and limitations of the proposed approach.

7.3.1 Methodological risks

As a criticism to the evaluation of the approach based on the two case studies, one may argue that both JHotDraw and Eclipse RCP have similar domains. This fact could then raise the question if the scope of applicability of DORIs is restricted only to GUI-related domains. As shown in both Table **??** and Table 7.2, the sets of concepts of both frameworks contain cases for every framework adaptation mechanism as well as for several combinations of them. Moreover, the domain variability models represented for both frameworks involve all the constructs of conceptual modeling. Therefore, assuming that frameworks rely on the given adaptation mechanisms, it seems reasonable to conclude that DORIs have a reasonable scope of applicability. However, it can only be actually claimed that the proposed approach has no limitations with respect to the framework's domain by developing DORIs for more frameworks, which address other domains.

The case studies were carried out by the author of this dissertation. The author is naturally an expert in the approach he proposes, and therefore, this non-neutral character of the evaluation also consists of a methodological risk.

7.3.2 Limitations

Without a supporting methodology, a framework developer may take some time to master the development of DORI modules, due to their different design style. Although the patterns given in Chapter 4 may help on this issue, they do not make it trivial.

Despite the learning issues, the main disadvantage of our approach is related to flexibility. The mechanisms to represent the meta-model elements in the DORI are not very flexible. Each modeling construct has a single way of being expressed. For instance, a concept must be represented in a module and the attributes must be represented in a constructor of that module. Although different ways to represent the same modeling construct could be considered, no practical significance in doing so has been found so far, while more constructs could compromise simplicity. Perhaps when applying the approach on more frameworks, this option could be revised if DORI modules are found "inelegant". In conventional generative approaches, classes, attributes, etc, can be mapped freely, in a sense that it is up to framework developers to decide how to map modeling elements to implementation elements. Therefore, the uniform representation of modeling constructs in DORI modules compromises flexibility to a certain extent.

Certain frameworks are built in such a way that their instantiation relies on external artifacts other than code written in the object-oriented language in which the framework is implemented. For instance, XML is a typical format for such external artifacts. In these cases, specialization aspects are not an appropriate option or they do not solve a significant part of the problem. For instance, frameworks such as Spring (Spring Source, 2008) and JBoss AOP (JBoss, 2008) rely heavily on XML. However, there is an important issue here. Such XML files have a similar role to a DSL, and can in fact be considered as such. An XML file (or a set of them) is representing an application model, which can be validated against a grammar (e.g. XML Schema) that plays a role similar to a meta-model. Therefore, up to a certain extent, an XML-based solution for using a framework and specialization aspects overlap in their goal of providing a means for higher level framework usage. This fact leads to the conclusion that, at least conceptually, the two kinds of usage cannot be easily combined.

Chapter 8

Related Work

This chapter compares the work of this dissertation with related approaches, which are divided into three main subjects. Section 8.1 addresses other approaches that combine frameworks and aspects. Section 8.2 addresses *feature-oriented programming* (FOP), a programming paradigm that handles features as first-class entities in the implementation of software systems. Finally, Section 8.3 addresses *domain-specific modeling* (DSM), the trend which the work in this dissertation follows.

8.1 Combining Frameworks and Aspects

Kulesza *et al.* (2006a) propose an approach for combining aspects with object-oriented frameworks. They propose the variability within the framework to be handled by *extension aspects*. Such aspects are based on the notion of *extension join points* (EJPs), which are locations that are exposed for introducing functionality. Aspects can introduce optional or alternative features at the EJPs. Such aspects introduce functionality at the framework classes, rather supporting the instantiation and composition of classes in a framework-based application. EJPs also serve the purpose of integrating the framework with other frameworks or APIs.

In the work of this dissertation, the aspects are the building blocks of a framework-based application, while they do not introduce functionality in the framework classes. Moreover, it is not meant that specialization aspects embody extra framework functionality. Specialization aspects simply form a higher-level reuse interface for building a framework-based application.

The use of extension aspects is an alternative to FOP (discussed in the next section), but following a more conservative approach. Extension aspects can be developed for existing frameworks by making lightweight modifications on them, while FOP approaches are revolutionary in the sense that they require a system to be structured in a significantly different manner. Contrarily to extension aspects and FOP, specialization aspects can be developed for an existing framework without modifying it.

There is nothing that prevents both extension aspects and specialization aspects to be used in combination. Extension aspects act solely on the framework classes, whereas specialization aspects act solely in framework-based application modules. However, combining both approaches arbitrarily would probably not be optimal. The point is that the framework design when using extension aspects for optional or alternative features is likely to be different than a framework design that enables the same variability by conventional instantiation. Therefore, it would make sense to combine both approaches if the variable parts that are addressed by extension aspects and by specialization aspects do not overlap, and accordingly, the framework is designed taking this separation into account.

Anastasopoulos & Muthig (2004) present an evaluation of AOP as a product-line implementation technology. The authors detail a case study on an hypothetical product-line using AspectJ, where aspects implement optional or alternative features, as in the case of Kulesza *et al.* (2006a).

Both approaches (Anastasopoulos & Muthig, 2004; Kulesza *et al.*, 2006b) have been used in combination with feature models, where feature configurations are the only input for obtaining system variants. However, as discussed in the next section, feature models impose certain limitations.

This dissertation demonstrates how AspectJ is suitable for implementing specialization aspects for Java frameworks. In the case of C++ frameworks, specialization aspects could be implemented using AspectC++ (Spinczyk *et al.*, 2002).

8.2 Feature-Oriented Programming (FOP)

In the Feature Oriented Domain Analysis (FODA) method, Kang *et al.* (1990) define a *feature* as "any prominent and distinctive aspect or characteristic that is visible to various stakeholders (i.e. end-users, domain experts, developers, etc)." Subsection 8.2.1 explains the concept of *feature models*, which are intrinsically related to FOP, Subsection 8.2.2 addresses FOP languages, and Subsection 8.2.3 discusses these subjects in the context of this dissertation.

8.2.1 Feature models

A *feature model* explicitly describes the possible variants of a system in terms of its features. A *feature configuration* is a selection of features of a feature model, which describes the features of a system variant. The initial proposal of feature models was given in the FODA method, while several extensions and refinements of it have been proposed throughout the years by other authors (e.g. Czarnecki & Eisenecker, 2000; Czarnecki *et al.*, 2005; Griss *et al.*, 1998; Gurp *et al.*, 2001). Despite some details in modeling constructs and/or notations, all the proposals have in common the following characteristics:

- The main entities are the features, which are organized hierarchically in a tree, where the tree root represents the system as a whole.
- Child features, i.e. which are nested under a parent feature, may only be part of a system if its parent feature also is.
- There is a distinction between optional and mandatory features.

Figure 8.1 presents an example of a feature model, using domain concepts similar to those given throughout the examples of this dissertation. The root concept is *application*. The filled circle denotes a mandatory feature (e.g. *menus*), whereas the non-filled circle denotes an optional feature (e.g. *toolbar*). Feature groups, denoted by connecting the parent-child links of features, may have configuration constraints. For instance, the group formed by *file*, *edit*, and *help* defines a constraint that enforces that one up to three features of the group have to be selected. As another example, the *default* and *user-defined* are alternative features.

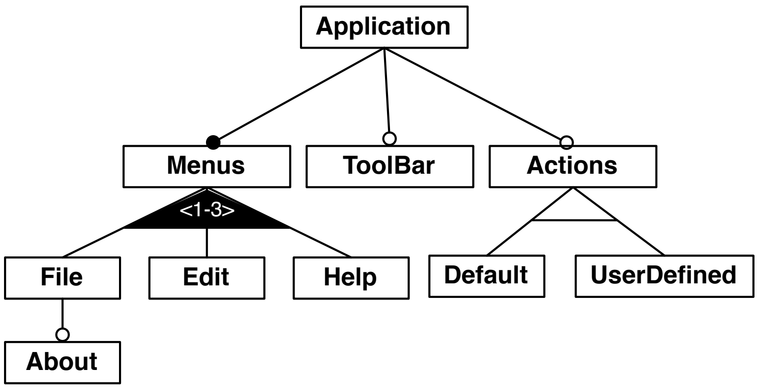


Figure 8.1: Example of a feature model.

Therefore, by reading the diagram one can infer that an application must have menus, which can be the file, edit, and help menus. At least one of these menus has to be part of an application. The file menu can have an optional *about* item. The toolbar and the *actions* are optional features. The latter are either default or user-defined.

Figure 8.2 presents a valid configuration of the features of the model given in Figure 8.1. The selected features are represented in gray, and define an application with the file menu, which has the about action, with the help menu, and that uses default actions.

Feature models may not be sufficient for expressing variants of a system, due to their limitations with respect to expressiveness (Czarnecki, 2004). A feature

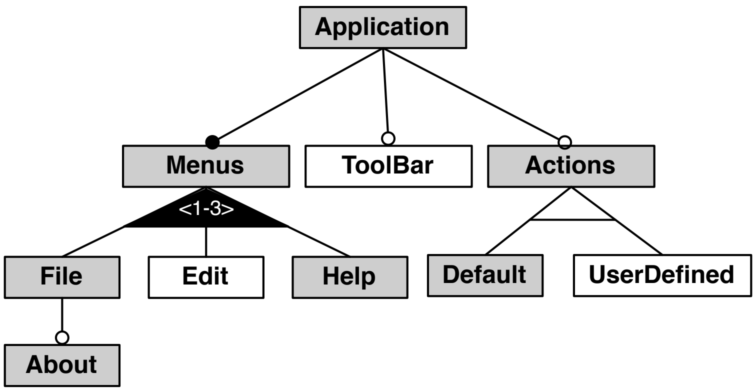


Figure 8.2: Valid feature configuration of the feature model of Figure 8.1.

model represents a finite variability space where every system variant is anticipated. If there is need for more expressive descriptions of system variants, other means such as conceptual models become necessary.

Although more expressive means for describing system variants might be necessary, feature models can still be used as a first step for expressing a system variant. Czarnecki & Antkiewicz (2005) propose an approach to map features to other modeling artifacts, such as class models, so that the latter can vary according to feature configurations. Class models may be representing domain variability models such as the ones proposed in this dissertation, enabling them to be adaptable according to feature configurations.

As an example of a limitation in the expressiveness of feature models, consider that in the given example, we would like to have the *about* item as an optional feature that could be part of any of the menus. A feature configuration would then define in which menu the about item would be. A possible option could be to have an optional *about* feature as a child of each menu (i.e. file, edit, help). However, this option would introduce redundancy and cluttering in the feature model. Next subsection explains related problems at the implementation level.

8.2.2 FOP languages

FOP is a programming paradigm where features are a first-class entity in a system's implementation. In FOP languages, such as *AHEAD* (Batory *et al.*, 2003), *CaesarJ* (Mezini & Ostermann, 2004), *aspectual mixin layers (AML)* (Apel *et al.*, 2006), or *ClassBox/J* (Bergel *et al.*, 2005), a system can be structured by means of several *feature refinements*. Each refinement cohesively implements a feature by refining a set of classes of the system. By cohesively, it is meant that a refinement completely implements the associated feature in the system. If we consider the general concept of feature models as presented in Subsection 8.2.1, FOP approaches enable a one-to-one mapping between the features of a feature model and the refinements which implement them. In this way, with a feature configuration, one obtains a system variant solely by *synthesizing* the refinements according to the selected features. Such a process is automated by a tool or a compiler. Although the authors of ClassBox/J do not characterize their work as a FOP language, feature refinements can be implemented in ClassBox/J in an equivalent way to the other languages that were mentioned.

Along with the limitations in the expressiveness of feature models, systems built using FOP also have limitations with respect to variability, namely due to the lack of mechanisms that enable an elegant implementation of all possible system variants without duplicated code. Moreover, FOP languages are not adequate for addressing an infinite variability space of system variants, where all the possible compositions of system elements cannot be anticipated.

In order to illustrate the limitations of FOP, a simple hypothetical case of implementing the features given on Figure 8.1 is presented next. The syntax of AHEAD is used, and only the features *application*, *menus*, *file*, and *about*, are shown. The following would be the base feature of the system (Application).

```
feature Application;

class BaseApplication {
   // ...
   void createMenus(MenuBar mb) {
   }
}
```

BaseApplication is a class module with an empty method for creating the menus. Given that the menus can vary among system variants, it is up to feature refinements to complete the method. The following is a feature refinement (File) for including the file menu.

```
feature File;

class FileMenu extends Menu {
   FileMenu() {
   super("File");
      // ...
   }
}

refines class BaseApplication {
   void createMenus(MenuBar mb) {
      super.createMenus(mb);
      Menu fileMenu = new FileMenu();
```

```
        mb.add(fileMenu);
    }
}
```

The class FileMenu that extends Menu (suppose it is the system class that represents a menu) is defined, and the method createMenus(..) of the class BaseApplication (shown in the previous code snippet) is refined. Additionally to the existing behavior (super.createMenus(..)), the file menu is added to the MenuBar parameter.

Considering now the *about* item in the *file menu*, the following is a refinement (About) for addressing this feature.

```
feature About;

refines class FileMenu {
    refines FileMenu() {
        super();
        add(new About());
    }
}
```

The constructor of the class FileMenu (given in the previous code snippet) is refined for plugging in an instance of About, which represents an about item and can be plugged in using Menu.add().

The class BaseApplication if composed with the feature refinements File and About would result in a synthesized system variant having the *file menu* with the *about* item. However, coming back to the case given in Subsection 8.2.1, if we want to enable the about item to be plugged in any of the menus, there are problems not only with respect to the feature model, but also concerning the implementation. In this case, we would need different feature refinements for each menu, given that the plugging of the about item cannot be generalized for every menu. This solution is not good given that all these refinements would contain a lot of redundancy, and a new menu feature would require another refinement for handling the about item. Moreover, the problem would become more serious if other existing features would interact with the about item. If the about item is represented in several different refinements, the interacting feature must be represented in several refinements, one for each case of the about item.

8.2.3 Discussion

Although it might not always be the case, the set of possible applications based on a framework is often unbound and cannot be anticipated. In these cases, the possible compositions of framework-provided elements cannot be expressed as a feature model, and accordingly, a framework-based application cannot be expressed with detail just by having a feature configuration. More expressive constructs for feature models have been proposed, such as attributes and cardinalities (Czarnecki *et al.*, 2005). However, the use of these constructs is still not expressive enough for describing framework-based applications, and nevertheless, they cannot be seamlessly integrated with the constructs available in existing FOP languages.

In the context of the work of this dissertation, domain variability models assume the role of a feature model, whereas application models assume the role of feature configurations. However, as explained before, feature models are less expressive than conceptual models. A feature model can be represented in an equivalent conceptual model, while not every conceptual model can be represented in a feature model.

Figure 8.3 presents a conceptual model describing an equivalent variability to the feature model of Figure 8.1. However, the *about* feature was considered as part of any *menu*, according to the variability modeling problem introduced previously, in order to illustrate how this kind of variability can be represented in a more elegant way than if using a feature model. Moreover, the solution given in the conceptual model enables new *menus* to be added, while not requiring other changes in the model. This would not be possible with the given feature model.

Although the graphical notation of feature models might describe the features more naturally and explicitly, the possible instances of the conceptual model are equivalent to the possible feature configurations of the feature model. Figure 8.4 represents an instance of the conceptual model of Figure 8.3 that represents the same feature configuration as in Figure 8.2.

In order to illustrate the different expressiveness of feature models, consider for instance the simple conceptual model given in Figure 5.4, which was used as an example for introducing DORIs. Such a conceptual model cannot be expressed through a feature model. Therefore, there is a clear difference between the kind of variability management enabled by what is proposed in this dissertation, i.e. using conceptual models and frameworks, and the variability based on feature models and FOP. The possible feature configurations of a feature model correspond to a finite set of trees, while the possible instances of a conceptual model can correspond to an infinite set of directed graphs. A tree is a constrained type of graph, which explains the different expressiveness of feature models.

Despite the different expressive power, FOP and framework specialization aspects have a commonality in their strategy, which is to raise the abstraction level of the solution space, so that the mapping of problem space abstractions becomes straightforward. However, in a system built using FOP, the feature refinements are simply part of the system and can be used just by including them in a synthesized system, without the need of developing additional code. On the other hand, framework specialization aspects imply the development of application aspects for including the features. This is due to the fact that FOP features are simply part of a system or not, whereas an application aspect is part of a framework-based application in a certain context and with certain parameters, which have to be given by the aspect.

From the point of view of feature cohesion, the application aspects have a nature similar to the feature refinements of FOP systems. In both cases, each module cohesively implements a feature in an application, which can be included/excluded by including/excluding the module. The difference is that a feature refinement affects the system modules and it is applied equally in all system variants, whereas each application aspect only modifies the modules of a framework-

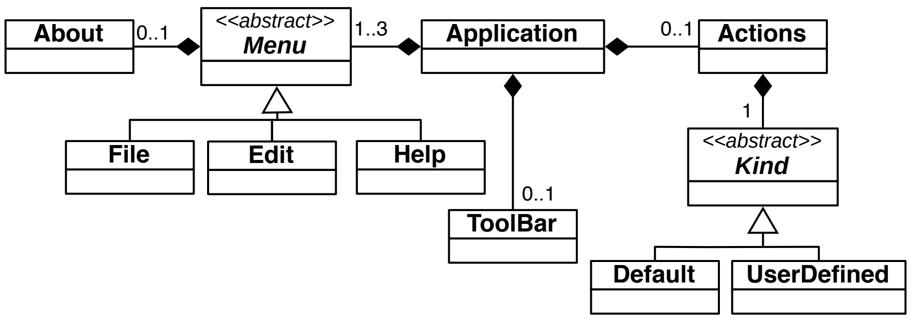


Figure 8.3: Feature model of Figure 8.1 represented as a conceptual model.

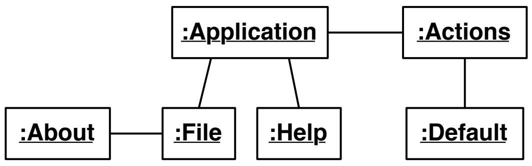


Figure 8.4: Instance of the classes of Figure 8.3, representing the feature configuration given in Figure 8.2.

based application in a certain context. Classbox/J is an exception to the other FOP languages, given that refinements take effect within a unit of scoping (a *classbox*), enabling to have several versions of a same class coexisting in the system.

Batory *et al.* (2000) argue that frameworks have limitations concerning inclusion/exclusion of optional features as a basis for the motivation for FOP, but a comparison of the variability enabled by the two approaches is not available. FOP is a revolutionary approach, given that although it is built upon object-orientation it requires a system to be refactored into feature refinements. On the other hand, framework specialization aspects are evolutionary, given that they assume frameworks as they are currently built and do not require any modifications on their implementation. An advantage of such an evolutionary approach is that it does not “dismiss” the solid body of existing knowledge on framework construction. Frameworks can be built as they have been built for years, and additionally, specialization aspects can be developed to support their usage on a higher level.

Finally, as well as in the case of *extension aspects* (previous subsection), apparently there is no obstacle to having a framework where feature refinements modularize optional and alternative features, while specialization aspects give support for adapting other variable parts in framework-based applications that are not simply about inclusion/exclusion of predefined features.

8.3 Domain-Specific Modeling (DSM)

Domain-specific modeling (DSM) (DSM Forum, 2007) is a trend that promotes a software development paradigm where applications of a certain domain are generated from domain-level descriptions, often referred to as *domain-specific modeling languages* (DSMLs). MetaEdit+ (MetaCase, 2008), Microsoft DSL Tools (Greenfield & Short., 2005), Generic Modeling Environment (GME) (Ledeczi *et al.*, 2001), or Eclipse-based technologies (EMF, JET, GMF Eclipse Foundation, 2007c), are examples of language workbenches that support the development of DSM solutions, providing the infrastructure for defining meta-models, graphical concrete syntax for models, and code generators.

A particular and popular application case of DSM is generation of framework-based applications — the case which this dissertation concentrates on. Such kind of DSM approach has been in practice for long. Roberts & Johnson (1997) used the term *visual builder* for referring to a DSML, and point out that black-box frameworks are suitable for building such a solution. The work in this dissertation goes along this trend, in the sense that the specialization aspects form a higher-level black-box reuse interface.

In conventional DSM approaches, meta-models and code generators are developed independently from the framework. Framework developers have to ensure the consistency between the framework, the meta-model, and the code generator. As the domain evolves, these three elements have to evolve accordingly. Experience reports reveal that the code generator should be kept as simple and straightforward as possible, in order to avoid difficulties related with domain evolution (Pohjonen & Tolvanen, 2002).

This dissertation proposes a DSM approach that dissents from the state-of-the-practice, where DSMLs are encoded in the solution space, rather than as a separate abstraction in the problem space. As explained, this is possible by raising the abstraction level of the solution space (i.e. the framework plus the specialization aspects). The DSM philosophy (DSM Forum, 2007) is in favor of having DSML definitions which follow a *domain-oriented* approach, rather than an *implementation-oriented* approach, so that the languages support modeling at higher abstraction levels. The work presented in this dissertation also sticks to this philosophy. Given that DSMLs are proposed to be defined in the implementation elements, one could associate this to an implementation-oriented approach. However, the DSML is not defined in the conventional reuse interface, but it is instead defined on the DORI modules, which constitute a higher-level reuse interface whose extensions closely resemble application concepts and their relationships.

Although the proposed approach does not require implementing code generators and defining DSMLs separately, it can be applied using the existing language workbenches, using their meta-model format and facilities for defining the concrete syntax of a DSML. Therefore, the adoption of DORIs is not an alternative to existing language workbenches, it simply proposes a way to develop an "enhanced" solution space. In case one does not want to use the generic code generator for any particular reason, code generators can still be defined manually — a task that becomes facilitated due to the higher-level reuse interface based on specialization aspects.

Antkiewicz & Czarnecki (2006) present the idea of having Framework-Specific Modeling Languages (FSMLs) which support round-trip engineering. As happens in our approach, code generators do not have to be developed. The FSMLs are defined through feature models, and code generation relies on mappings between features and framework elements. The use of a FSML supports a development paradigm in which an application is obtained by generating code that is meant to be completed manually. Given that FSMLs are defined in the form of feature models, the definition of the variability space has similar constraints to systems built using FOP. However, the authors present the approach as appropriate for solutions where non-generated code has to be developed manually.

The approach of this dissertation does not intend to support round-trip engineering. Instead, a mechanism for clear separation between generated and manually written code is adopted. Following the DSM philosophy of raising the abstraction level by hiding complexity, generated code is not intended to be manipulated or understood.

Due to the reasons pointed out in Section 7.3.2, conventional DSM approaches, as well as FSMLs, are more flexible with respect to the mapping of concepts to framework-based code. While DORIs are less flexible, there are automation gains and the DSM solution relies only on the framework implementation.

Chapter 9

Conclusions and Future Work

This chapter concludes the dissertation by summarizing the contributions (Section 9.1), outlining future work (Section 9.2), and presenting some final remarks (Section 9.3).

9.1 Summary of Contributions

The essence of the contributions of this dissertation is twofold. This work proposes:

- An effective technique for building framework reuse interfaces using *specialization aspects*, which enable the development of framework-based applications at a higher abstraction level than with conventional reuse interfaces. In this approach, a framework-based application is composed of several *application aspects* that are based on the specialization aspects.

- A technique for encoding the definition of a conceptual model within a reuse interface composed of specialization aspects, so that the transformation of instances of that conceptual model into application aspects can be generalized. The technique relies on the close relation between application aspects and application models, and on the uniform mechanism to instantiate concepts. The proposed kind of reuse interface was named *domain-oriented reuse interface* (DORI).

The DORI concept gave the title of this dissertation, due to the belief that it is the ultimate end of what is proposed, while specialization aspects are a means for achieving it. The ALFAMA language workbench that was developed, together with the case studies using two real frameworks, stands as a proof-of-concept that the proposed approach is feasible in realistic settings. The approach is evolutionary, given that it is applicable to frameworks as they have been built already for many years, and therefore, all the solid knowledge on framework construction as well as the existing frameworks are not dismissed.

9.2 Future Work

This section outlines several topics for future research work that can be carried out in sequence of what was proposed in this dissertation.

9.2.1 Constraints on DSMLs

As explained in Chapter 5, a DORI is capable of encoding a *domain-specific modeling language* (DSML), whose definition is given in terms of a conceptual model. Such a conceptual model comprises concepts with attributes, abstract concepts, concept inheritance, composite and directed associations. Although the modeling constructs are fairly rich, detailed domain rules may require more expressive means to enforce them. Such detailed constraints can be specified on top of the conceptual model using an additional language, such as the Object Constraint Language (OCL) (OMG, 2003). In doing so, the possible instances of the conceptual model are constrained by OCL expressions, which enforce more detailed domain rules. Scenarios using OCL (or a similar means) were not experimented in the case studies that were carried out. New annotation types on DORI modules could support the definition of these constraints.

A DSML covers the features of a given framework. If there are a large number of features, it is likely that the number of DSML concepts is also large. Given this, another kind of constraint for the DSML that would be useful is related to its adaptability according to yet more high-level abstractions, which could be for instance configurations of a feature model. A feature model could represent high-level features, which by themselves cannot completely specify an application, but which could constrain the DSML according to the features that one wants to use in an application model. This could be achieved by associating features with sets of DSML concepts. A feature configuration would thus be a first step for describing an application, and then one would instantiate the adapted DSML. For instance, a feature "Menus" in a feature model could be associated with the set of concepts of the DSML "Menu", "Submenu", and "Menu Action", so that these concepts would not be available in the DSML if the feature would not be selected. In a large-scale setting, reducing the size of the DSML using high-level feature abstractions, could facilitate the tasks of learning a DSML and maintaining application models. Moreover, the usability of modeling environments could be improved, given that there would be fewer interaction options due to the reduced number of DSML concepts.

9.2.2 Concrete syntax of DSMLs

This dissertation did not address the definition of a concrete syntax for DSMLs. However, although this issue is orthogonal to the proposed work and can be supported independently, it is of extreme importance. Graphical concrete syntax is attractive for describing application models. Graphics are typically capable of offering better usability than text-based concrete syntax. Moreover, a graphical application model can assume a double role. It is simultaneously the formal specification of the application and a significant part of its documentation, given that graphics are typically used for documenting applications. Graphical descriptions are essentially based on *nodes*, and *links* that connect those nodes. Such elements are represented in figures, which are typically related to existing domain notations. Having an abstract syntax (e.g. defined as a meta-model), the information for associating its elements to nodes and figures has to be given elsewhere. Future work could explore the feasibility of having a way to also specify within a DORI the mapping of elements to nodes and links. Additional tool support could then use this information as input for generating a graphical modeling environment.

9.2.3 Integration of Frameworks and their DSMLs

Framework integration is a well-known problem (Fayad *et al.*, 1999). Apparently, there is no obstacle for using specialization aspects for integrating applications based on two different frameworks. However, such scenarios were not explored in the context of this dissertation. Assuming that this would be possible and that each of the frameworks to be integrated has a DSML derived from its DORI, an interesting direction would be to explore how to develop a DSML that integrates those DSMLs. It could be experimented if such DSMLs could rely on specialization aspects that "bridge" the specialization aspects of the frameworks to be integrated.

9.3 Final Remarks

The author of this dissertation believes that the path that has to be followed for increasing productivity and quality in software development should heavily rely on *reuse* and on *raising the level of abstraction* of development artifacts, rather than relying on more sophisticated general-purpose programming languages.

Frederick P. Brooks (1987) classifies software complexity into *essential* and *accidental*. Essential complexity is inherent to the problem that the software solves, whereas accidental complexity arises due to the abstractions that are being used to implement the solution to that problem. More sophisticated programming languages may provide better abstractions for solving problems, but are not relieving developers from the essential complexity of those problems. Object-oriented frameworks tackle essential complexity, given that they provide an abstract solution for a family of related problems. A framework may be already addressing a great part of the essential complexity of a problem, while framework-based applications only have to fill-in missing parts which differentiate the problems. Here resides the power of frameworks.

Although frameworks are powerful, successful framework-based application development may not be easy to achieve. In the first place, building a framework is far more costly than building a normal application and requires good designers and domain experts. Enabling the "right" variability may be challenging, given that the framework has to meet the requirements of a typically wide set of applications. Moreover, the variability requirements are likely to evolve constantly, given the demands from multiple applications and the unavoidable domain evolution. Once the variability requirements are met, the usability of the framework considerably affects its effectiveness. A framework that is difficult to use may compromise the pay-off of having a framework-based development. Application developers have to understand the reuse interface of the framework, a task which may require a considerable investment even if the domain concepts are well-understood. Therefore, framework learning involves a significant amount of accidental complexity, given that application developers are faced with a difficulty that is not related with the essence of the problem.

In order to reduce the accidental complexity of framework learning, several usage strategies have been proposed, as explained in Chapter 2. On such proposals, all the kinds of external artifacts to the framework have a common goal, which is to assist the use of the solution space abstractions (i.e. framework), and thus, tackling accidental complexity. F-DSLs have a close to ideal usability in terms of framework usage support. Application developers only need to understand about domain concepts and cope with a language with very little accidental complexity, given that, ideally, an F-DSL solely deals with the essential complexity of the problem domain.

Conventional approaches for building F-DSLs are based on creating independent abstractions on the problem space, which are mapped to the solution space abstractions. Although the F-DSLs reduce accidental complexity in application development, its implementation and maintenance have a fairly significant amount of both essential and accidental complexity. The essential complexity is associated with the mapping complexity and its constant evolution. Having the F-DSL concepts set, no matter which technology is adopted, the essential complexity will be the same given that the mapping is equally complex in all solutions. A particular technology may help to cope with the accidental complexity associated with mapping primitives, consistency, and debugging, but it will never simplify the mapping itself.

This dissertation went in the direction of improving the abstractions for defining the solution space, rather than relying on problem space abstractions. The higher-level framework reuse interfaces using specialization aspects reduce the essential complexity of the mapping between concepts and code. Going one step further, the proposed language workbench reduces the accidental complexity of implementing transformations, given that code generators do not have to be developed.

Given that DORIs and the proposed language workbench reduce essential and accidental complexity accordingly, how can one demonstrate that the complexity of developing DORIs does not outweigh the complexity of conventionally developing an F-DSL? The question is indeed pertinent, and logically, the author of this dissertation favors DORIs. One could think that this could only be properly evaluated with quantitative data gathered from empirical studies which compare the development of DORIs and conventional solutions. However, doing this in an unquestionable manner would be too costly, if even feasible. This dissertation made an attempt to present evidence to the reader that DORIs are less complex than conventional solutions, by comparing several illustrative cases. Such evidence did not rely on toy examples, but it rather involved the use of real frameworks, as described in the case studies, and a working prototype tool.

References

AGUIAR, A. (2003). *Framework documentation, a minimalist approach*. Ph.D. thesis, Faculty of Engineering, University of Porto.

ALEXANDER, C., ISHIKAWA, S., SILVERSTEIN, M., JACOBSON, M., FIKSDAHL-KING, I. & ANGEL, S. (1977). *A Pattern Language — Towns, Buildings, Construction.*. Oxford University Press.

ANASTASOPOULOS, M. & MUTHIG, D. (2004). An evaluation of aspect-oriented programming as a product line implementation technology. In *ICSR'04: Proceedings of the 8th International Conference on Software Reuse*.

ANTKIEWICZ, M. & CZARNECKI, K. (2006). Framework-specific modeling languages with round-trip engineering. In *MoDELS'06: Proceedings of the 9th International Conference Model Driven Engineering Languages and Systems*.

APEL, S., LEICH, T. & SAAKE, G. (2006). Aspectual mixin layers: aspects and features in concert. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*.

BATORY, D., CARDONE, R. & SMARAGDAKIS, Y. (2000). Object-oriented frameworks and product lines. In *Proceedings of the 1st Software Product Line Conference*.

BATORY, D., SARVELA, J.N. & RAUSCHMAYER, A. (2003). Scaling step-wise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*.

Bergel, A., Ducasse, S. & Nierstrasz, O. (2005). Classbox/J: controlling the scope of change in java. In *OOPSLA '05: Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Bosch, J. (1999). Product-line architectures in industry: a case study. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*.

Bosch, J. (2000). *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co.

Cleaveland, J.C. (1988). Building application generators. *IEEE Software*, **5**, 25–33.

Colyer, A. & Clement, A. (2004). Large-scale AOSD for middleware. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*.

Czarnecki, K. (2004). Overview of generative software development. In *Proceedings of the International Workshop on Unconventional Programming Paradigms*.

Czarnecki, K. & Antkiewicz, M. (2005). Mapping features to models: a template approach based on superimposed variants. In *GPCE'05: 4th International Conference Generative Programming and Component Engineering*.

Czarnecki, K. & Eisenecker, U.W. (2000). *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co.

Czarnecki, K., Helsen, S. & Eisenecker, U.W. (2005). Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, **10**, 7–29.

DSM Forum (2007). Workshops on domain-specific modeling, 2001-2006. http://www.dsmforum.org/DSMworkshops.html.

Durham, A.M. & Johnson, R.E. (1996). A framework for run-time systems and its visual programming language. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 406–420.

Eclipse Foundation (2007a). AspectJ programming language. http://www.eclipse.org/aspectj.

Eclipse Foundation (2007b). Eclipse platform. http://www.eclipse.org.

Eclipse Foundation (2007c). Eclipse platform and projects. http://www.eclipse.org/projects.

Eclipse Foundation (2007d). EMF – Eclipse Modeling Framework. http://www.eclipse.org/emf.

Eclipse Foundation (2007e). GMF – Graphical Modeling Framework. http://www.eclipse.org/gmf.

Fairbanks, G., Garlan, D. & Scherlis, W. (2006). Design fragments make using frameworks easier. In *OOPSLA '06: Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*.

Fayad, M.E., Schmidt, D.C. & Johnson, R.E. (1999). *Building application frameworks: object-oriented foundations of framework design*. John Wiley & Sons, Inc.

Filman, R.E. & Friedman, D.P. (2004). Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*, chap. 2, 21–35, Addison-Wesley.

Fowler, M. (2008). Martin Fowler's Bliki. http://www.martinfowler.com/bliki/.

Frederick P. Brooks, J. (1987). No silver bullet: essence and accidents of software engineering. *Computer*, **20**, 10–19.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc.

Greenfield, J. & Short., K. (2005). *Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools.*. John Wiley and Sons.

Griss, M.L., Favaro, J. & d' Alessandro, M. (1998). Integrating feature modeling with the RSEB. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*.

Gurp, J.V., Bosch, J. & Svahnberg, M. (2001). On the notion of variability in software product lines. In *WICSA'01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture*.

Hakala, M., Hautamäki, J., Koskimies, K., Paakki, J., Viljamaa, A. & Viljamaa, J. (2001). Architecture-oriented programming using FRED. In *ICSE'01: Proceedings of the 23rd International Conference on Software Engineering (Formal research demo)*.

Hanenberg, S., Schmidmeier, A. & Unland, R. (2003). AspectJ idioms for aspect-oriented software construction. In *EuroPLOP'03: 8th European Conference on Pattern Languages of Programs*.

Hautamäki, J. & Koskimies, K. (2006). Finding and documenting the specialization interface of an application framework. *Software: Practice and Experience*, **36**, 1443–1465.

Holmes, R., Walker, R.J. & Murphy, G.C. (2005). Strathcona example recommendation tool. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

JBoss (2008). JBoss AOP. http://www.jboss.org/jbossaop/.

Johnson, R.E. (1992). Documenting frameworks using patterns. In *OOPSLA '92: Proceedings of the 7th ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications*.

Johnson, R.E. (1997). Frameworks = (components + patterns). *Commun. ACM*, **40**, 39–42.

Johnson, R.E. & Foote, B. (1988). Designing reusable classes. *Journal of Object-Oriented Programming*, **1**, 22–35.

Johnson, S.C. (1979). Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, Holt, Rinehart, and Winston.

Kang, K., Cohen, S., Hess, J., Nowak, W. & Peterson, S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Tech. rep., Carnegie Mellon University.

Kelly, S. & Tolvanen, J.P. (2008). *Domain-Specific Modeling*. Wiley-IEEE Computer Society Press.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M. & Irwin, J. (1997). Aspect-Oriented Programming. In *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*.

Krasner, G.E. & Pope, S.T. (1988). A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, **11**, 26–49.

Kulesza, U., Alves, V., Garcia, A.F., de Lucena, C.J.P. & Borba, P. (2006a). Improving extensibility of object-oriented frameworks with aspect-oriented programming. In *ICSR'06: Proceedings of the 9th International Conference on Software Reuse*.

Kulesza, U., Lucena, C., Alencar, P.S.C. & Garcia, A. (2006b). Customizing aspect-oriented variabilities using generative techniques. In *SEKE'06: Proceedings of the 18th International Conference on Software Engineering & Knowledge Engineering*.

Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garett, J., Thomason, C., Nordstrom, G., Sprinkle, J. & Volgyesi, P. (2001). The generic modeling environment. In *WISP'01: Proceedings of the Workshop on Intelligent Signal Processing*.

McAffer, J. & Lemieux, J.M. (2005). *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley Professional.

MetaCase (2008). MetaEdit+ tool. http://www.metacase.com.

Mezini, M. & Ostermann, K. (2004). Variability management with feature-oriented programming and aspects. In *ACM Conference on Foundations of Software Engineering (FSE-12)*.

Moser, S. & Nierstrasz, O. (1996). The effect of object-oriented frameworks on developer productivity. *Computer*, **29**, 45–51.

OMG (2003). *UML 2.0 OCL Specification*. OMG.

OMG (2004). *UML Superstructure Specification, v2.0*.

OMG (2006). MOF 2.0 QVT : Queries / Views / Transformations. http://www.omg.org.

Ortigosa, A., Campo, M. & Moriyón, R. (2000). Towards agent-oriented assistance for framework instantiation. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Pohjonen, R. & Tolvanen, J.P. (2002). Automated production of family members: Lessons learned. In *PLEES'02: 2nd International Workshop on Product Line Engineering - The Early Steps: Planning, Modeling, and Managing*.

Pohl, K., Böckle, G. & van der Linden, F.J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc.

Pree, W. (1995). *Design patterns for object-oriented software development*. ACM Press/Addison-Wesley Publishing Co.

Roberts, D. & Johnson, R.E. (1997). Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Pattern Languages of Program Design 3*, Addison Wesley.

Santos, A.L. (2007). Automatic support for model-driven specialization of object-oriented frameworks using ALFAMA. OOPSLA'07 Demonstrations Track.

Santos, A.L. (2008). ALFAMA: Automatic DSLs for using Frameworks by combining Aspect-oriented and Meta-modeling Approaches. AOSD'08 Demonstrations Track.

Santos, A.L. & Koskimies, K. (2006). Aspects as specialization units for framework-based SPLs. BPAOSD'06: Workshop on Best Practices in Applying Aspect-Oriented Software Development (AOSD'06).

Santos, A.L. & Koskimies, K. (2008). Modular hot spots: A pattern language for developing high-level framework reuse interfaces. In *EuroPLOP'08: 12th European Conference on Pattern Languages of Programs*.

Santos, A.L., Lopes, A. & Koskimies, K. (2006). Modularizing framework hot spots using aspects. In *Proceedings of the 11th Spanish Conference on Software Engineering and Databases*.

Santos, A.L., Lopes, A. & Koskimies, K. (2007). Framework specialization aspects. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*.

Santos, A.L., Koskimies, K. & Lopes, A. (2008). Automatic domain-specific modeling languages for generating framework-based applications. In *SPLC '08: Proceedings of the 12th Software Product Lines Conference*.

SEI (2008). Software product lines. http://www.sei.cmu.edu/productlines/.

SourceForge (2006). JHotDraw framework. http://www.jhotdraw.org.

Spinczyk, O., Gal, A. & Schröder-Preikschat, W. (2002). AspectC++: An aspect-oriented extension to C++. In *Proceeding of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*.

Spring Source (2008). Spring Framework. http://springframework.org/.

Steyaert, P., Lucas, C., Mens, K. & D'Hondt, T. (1996). Reuse contracts: Managing the evolution of reusable assets. In *OOPSLA '96 Conference on Object-Oriented Programming Systems, Languagges and Applications*.

Tarr, P., Ossher, H., Stanley M. Sutton, J. & Harrison, W. (2004). N degrees of separation: Multi-dimensional separation of concerns. In *Aspect-Oriented Software Development*, chap. 3, 37–61, Addison-Wesley.

van Deursen, A., Klint, P. & Visser, J. (2000). Domain-Specific Languages: an annotated bibliography. *SIGPLAN Not.*, **35**, 26–36.

Viljamaa, J. (2003). Reverse engineering framework reuse interfaces. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

W3C (2008a). Extensible Markup Language (XML). http://www.w3.org/XML/.

W3C (2008b). XSL Transformations (XSLT). http://www.w3.org/TR/xslt.

Weinand, A., Gamma, E. & Marty, R. (1989). Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, **10**, 63–87.

Zelkowitz, M.V. & Wallace, D.R. (1998). Experimental models for validating computer technology. *Computer*, **31**, 23–31.