MARCUS FELIPE MONTENEGRO CARVALHO DA FONTOURA

A SYSTEMATIC APPROACH TO FRAMEWORK DEVELOPMENT

Dissertation presented to the Computer Science Department of the Pontifical Catholic University of Rio de Janeiro as part of the requirements for the fulfillment of the doctor's degree in Computer Science.

Advisor: Carlos José Pereira de Lucena

Computer Science Department
Pontifical Catholic University of Rio de Janeiro

Rio de Janeiro, July 5, 1999.

Abstract

This work proposes a design language that represents the frameworks hot-spots and its instantiation process as first-class citizens. The language is defined as an extension to UML. Knowledge-based transformational tools are described to illustrate how the proposed design language can be used to automate several steps of a framework development process: design, implementation, instantiation, and maintenance. The semantics of the language is formally presented to enhance its understanding by a wide variety of users and to allow for the construction of new supporting tools. A consequence of this formalization is a precise definition of what frameworks really are. Several case studies are presented to illustrate the use of the approach in real-world framework development scenarios.

Summary

List of Figures

Chapter 1 – Introduction

This work proposes a design language for framework specification and shows how it can be used to automate several steps in a framework development process. This introduction describes some problems related to framework development and presents a brief overview of the proposed solutions. Finally, an outline of how the thesis dissertation is organized and a summary of the contributions are presented.

1.1 MOTIVATION

An object-oriented (OO) framework [27, 46] differs from a standard application because some of its parts, which are the hot-spots or flexible points [70, 88], may have a different implementation for each framework instance, and are left incomplete until instantiation time. A framework models the behavior of a family of applications [68]. Its kernel represents the similarities among the applications and the specific application behavior is provided by the hot-spots. The framework development process differs from the development of standard applications since an extra phase is necessary to complete the hot-spots, providing the specific behavior needed by the framework instances. This new phase is the instantiation phase.

Although various techniques and combination of techniques have been proposed to support object-oriented design [17] none of them provides appropriate constructs to describe the framework's hot-spots. These techniques simply neglect the fact that frameworks hot-spots need to be represented differently from the rest of the system [71]. This work proposes the use of a new design language to represent frameworks and shows how it can help the systematization of the design, implementation, instantiation, and maintenance steps of a framework development process. The proposed framework design language is presented as an extension to UML [82]. The new language constructs are described through UML extensibility mechanisms: stereotypes, tagged values, and constraints.

The systematization approach is based on knowledge based transformational tools that can be applied to specific steps of the process. These tools have been implemented and tested for the development and analysis of several large real-world frameworks [5, 30, 31, 77, 96].

The semantics of the proposed UML-extended design language was formalized to facilitate its understanding by a wide variety of users and to allow for the construction of new supporting tools.

1.2 PROBLEM STATEMENT

Based on our experience in developing complex frameworks [30, 31, 77] and in teaching courses on frameworks [22] some conclusions could be put together:

1. Frameworks differ from standard applications since they need to be instantiated. The instantiation process consists of completing the framework hot-spots in order to create a fully functional application. Some hot-spots may require compile-time instantiation (the hot-spot has to be configured before the application is running) while others may require run-time instantiation (the missing information is completed only during run-time). If a framework has at least one compile-time hot-spot it is not yet a valid application, since it cannot run until the missing hot-spot information is provided;
2. Frameworks are normally represented through object-oriented analysis and design methods (OOADMs) diagrams, such as OMT [81] and UML [82] class and interaction diagrams;
3. Current OOADMs do not provide elements and diagrams to represent hot-spots, their semantics, and how they should be instantiated;
4. Generally, frameworks are complex design structures that may lead to very tangled diagrams.

The items 3 and 4 described above are problems related to *framework design representation* that have serious impacts in all the steps of the framework development process: design, implementation, instantiation, and maintenance, as described in the following subsections. Requirements elicitation and domain analysis [72] will not be covered by this work. However, representing frameworks through an appropriate design language forces the framework designer to focus on hot-spots at an earlier phase in the development process, and may be of assistance in the requirements elicitation/domain analysis phase.

1.2.1 FRAMEWORK DESIGN

The framework's hot-spots are not first-class citizens in current OO design languages. They have to be represented as a combination of standard object-oriented constructs, which leads to two undesirable consequences. The first is the difficulty of identifying the hot-spots in the framework structure, encumbering the instantiation process. The second is the complexity of the design diagrams, especially when "tricks" are used to add flexibility and extensibility to the framework, and overcome the lack of appropriate design elements. Some design patterns [13, 34, 95] are examples of "tricks" that might be used.

1.2.2 FRAMEWORK IMPLEMENTATION

The implementation of the framework hot-spots is one of the most critical parts in the framework development. Several techniques such as design patterns [13, 34, 95], meta-level programming [51], and contracts [39, 40] have been proposed to overcome the lack of appropriate constructs to model flexibility and extensibility mechanisms in current OO programming languages. However, the selection of the most appropriate technique for each of the framework's hot-spots may be a very difficult task. If the hot-spots and their properties are not explicitly represented, this task can become even harder.

1.2.3 FRAMEWORK INSTANTIATION

Framework usability is a problem noticed in companies that use this technology to create applications, in part because of the intrinsic complexity of the instantiation process but also due to the lack of appropriated documentation techniques and tools [16]. The most common way to instantiate a framework is to inherit from abstract classes defined in the framework hierarchy and write the code that is called by the framework itself. However, it is not always easy to identify, especially for non-expert users, which code is needed and where it should be written since framework class hierarchies can be very complex. Application developers have to rely on extra documentation to be able to create framework instances properly since it is very unlikely that they will be able to browse the framework code and write the required instantiation code if no appropriate documentation is provided.

Generally, framework instantiation is far more complex than simply plugging components into hot-spots. Hot-spots might have interdependencies [30, 31], might be optional [5, 31], frameworks may provide several ways of adding the same functionality [96], and so on. The instantiation process should be explicitly represented in an unambiguous form to allow application developers to create valid framework instances in a straightforward manner. This explicit process representation should also facilitate the construction of tools to assist the framework instantiation.

1.2.4 FRAMEWORK MAINTENANCE

Currently, there is very little tool support for framework maintenance and evolution. Some of the most common problems regarding framework evolution are described in [16] and summarized below:

- Structural complexity: the framework evolution can make its structure (object interfaces, class hierarchies, and so on) hard to manage and understand;
- Changes in the domain: as the framework evolves and new framework instances are created, new abstractions that should be part of the framework may be derived;
- New design insights: the framework design structure may need to be improved in the light of issues previously neglected or forgotten.

In all the three cases an explicit definition of the framework hot-spots may facilitate the job of the framework maintainer. Changing the implementation of an existing framework kernel class every time new functionality is needed, for example, is not good practice. If the hot-spot were appropriately marked, the framework and the application maintainers would know how they relate to the rest of the system and which methods and classes can be removed without disturbing the framework or the instantiated applications.

1.3 OVERVIEW OF THE PROPOSED SOLUTION

This work focuses on defining an appropriate framework design language and showing how it can help the systematization of some of the tasks in a framework development process.

The solution was based on extending the UML design language [65, 82] to enhance the representation of frameworks by: (i) representing the framework hot-spots differently from the framework kernel; (ii) classifying the hot-spots to represent their semantics; (iii) explicitly representing the framework instantiation; (iv) being abstract enough to simplify the diagrams and allow for several implementation approaches to be used. The UML extensibility mechanisms (stereotypes, tagged values, and constraints) were used to define the new language constructs.

Based on the proposed representation for frameworks, tools that automate the design, implementation, instantiation, and maintenance steps have been defined. However, the value of the proposed design language is independent from the tools described here, and different supporting tools may be defined later. The new language constructs are formally presented, allowing their semantics to be precisely understood by different users and tool authors.

The tools are presented as transformational systems [18, 83, 84] that modify the framework representation throughout the development life-cycle. The transformations are knowledge-based [84, 90, 91] and may require human interaction since several of their steps are undecidable (checking if two different implementations match, for example). All tools described in this dissertation are currently implemented in PROLOG. The tools described throughout this work are as follows:

- *Framework editor tool*: is responsible for storing the framework design representation in a knowledge-based form. It validates design syntax and semantics and supports several design analyses;
- *Code generation and documentation tools*: are responsible for generating the framework code and corresponding documentation from the framework design representation;
- *Process-based instantiation tool*: uses the instantiation process, defined as part of the framework design, to guide the framework instantiation. It transforms the framework into specific applications by completing the missing hot-spot information. This tool provides a compositional approach to framework instantiation [37], in which a process description guides how components are integrated into the framework;
- *Domain-specific language generator*: generates a domain-specific language (DSL) [41] from the framework design representation. The DSL helps the framework

instantiation by syntactically assuring the creation of valid framework instances. This tool provides a generative approach to framework instantiation [6, 37], in which the hot-spot missing information is specified as DSLs programs that are compiled to generate the framework instantiation code;

- *Maintenance tool*: is responsible for applying refactoring [47, 66] and unification rules [2], to the framework design specification. Refactorings are behavior-preserving transformations that may be applied on the framework design representation as well as on the framework implementation. Unifications are transformations that alter the framework design structure to incorporate new features or modify the existing ones;
- *Verifier tool:* may be used in any step of the development process to check the soundness of the transformations applied to the framework design representations .

1.4 ORGANIZATION OF THE THESIS

The next chapter provides background information. It defines a common vocabulary and describes current research in the areas related to this thesis: object-oriented design, frameworks and design patterns, transformational systems, knowledge-based software engineering (KBSE), and formal methods.

Chapter 3 presents several design approaches that are currently being used to model frameworks and introduces the proposed framework design language, describing its syntax and informal semantics.

Chapter 4 presents a transformational view of the framework development process, describing the artifacts and tools that are used in each of the steps: design, implementation, instantiation, and maintenance. This chapter focuses on how tools can help the systematization of the development process when frameworks are represented through the proposed design language.

Chapters 5, 6, 7, and 8 detail the tools described in chapter 4. Corresponding related work is presented in each of these chapters. The development process and the tools proposed in this dissertation serve to exemplify how the framework design language may be used to automate several development activities. However they are not unique, in the sense that different processes and tools that use the design language may be proposed later on.

Chapter 9 presents the formalization of the semantics of the framework design language. This formalization is useful for the understanding of the UML-extended concepts by a wide variety of users, who may be interested in simply using the language or in building different supporting tools. This chapter also shows how several transformations described throughout the dissertation may be formally verified based on the language semantics. A study of the computational complexity of the verification procedures is presented, showing that supporting tools may be constructed. The *verifier tool* is described as an example of such a tool.

Chapter 10 describes five case studies that show how the design language and its supporting tools can help the development and analysis of large real-world frameworks.

Chapter 11 concludes the thesis with a summary of the work presented throughout the dissertation and suggests directions for future research and improvements.

There are two appendixes: Appendix A summarizes the proposed design language while Appendix B describes the prototype implementation of its supporting tools.

1.5 STATEMENT OF CONTRIBUTIONS

The major contributions of this work are:

- Framework design language: the proposed UML-extended design language should be useful to represent frameworks more adequately than current techniques. This will be illustrated by showing that the systematization of several tasks in the framework development process can only be made if the framework hot-spots and instantiation process are explicitly represented. The idea behind the formalization of the language semantics is to enhance its usability and allow the definition of new supporting tools by many different users;
- Supporting tools: the tools presented here serve to confirm the usefulness of the proposed design language but should also be valuable on their own, illustrating possible ways of automating some of the tasks in a framework development process. Based on their description different tool authors can develop similar tools that best fit the needs of their organization and development methods. The *verifier tool* illustrates how the design language can be used as a basis for a rigorous software development

process, defining polynomial time procedures that verify the transformations applied to the software artifacts handled by the various development activities;

- Domain-specific languages (DSLs) and framework instantiation: although several authors have proposed the use of DSLs [41] to enhance framework usability [11, 37, 79], none of them have proposed a systematic approach to derive DSLs from the framework structure. This derivation is only possible due to the expressive power of the proposed design language;
- Study of several implementation and documentation techniques: the *code generation* and *documentation* tools are responsible for transforming the framework design representation into implementation level representation and generating appropriate documentation. These transformations are based on several implementation [34, 38, 50, 51, 62] and documentation techniques [32, 45, 52] that have been analyzed in the context of framework development;
- Unification rules: refactoring rules have already been successfully used to assist framework maintenance [47, 66, 78]. However, refactorings are behavior-preserving transformations and cannot, for example, deal with problems related to changes in the framework domain. The unification rules, proposed in this work, are complementary to the refactoring procedures and address some problems not supported by them;
- Formalization of the concept of frameworks: a consequence of the formal definition of the framework design language is the formal definition of what frameworks really are. Although they have been used by industry and academe for some time, until now there is no agreement on a common and formal definition of frameworks [46].

1.6 SUMMARY

Current OOADMs fail to represent frameworks appropriately since they do not represent the hot-spots and the framework instantiation as first-class citizens. This fact has implications in all the steps of the framework development process: requirements, design, implementation, instantiation, and maintenance. This work proposes an extension to the UML design language to overcome these problems and help the systematization of some of the framework development tasks. Knowledge-based transformation tools are proposed to illustrate how this systematization can be achieved. The semantics of the new design language is formally presented to enhance its usability by different users and allow for the construction of new supporting tools.

Chapter 2 – Background

This chapter defines the basic concepts used in this dissertation and describes the context of the work. Research in the field of object-oriented design is presented as a motivation for our work. Our view of design patterns and how they relate to frameworks is described. Finally, brief introductions about transformational systems, knowledge-based software engineering (KBSE), and formal methods are presented to enhance the understanding about the tools and the formalization approach used.

2.1 OBJECT-ORIENTED DESIGN

The increasing use of OOADMs [23, 74, 81, 82] can be attested by the UML standardization effort [65], to which several organizations and individuals have been contributing along a two year project. This growing interest in the area brought the attention of software researchers to the field of *object-oriented design.*

In the context of this dissertation, the term *object-oriented design* will be used to designate the research area that defines artifacts, processes, and tools to support the software development tasks that take place between the requirements elicitation and the implementation steps. Sometimes these tasks are divided into analysis and design tasks, but this division is irrelevant for the purposes of this work.

Although the UML standard is a great step towards a common basis for the field, there will always exist research efforts in the area, which include:

- The specification of new domain-specific or organization-specific elements or diagrams that may enhance the design languages for a specific task, organization, or project;
- The specification of new general-purpose elements or diagrams that may simplify or enhance the design and analyses tasks, clarify diagrams, allow new consistency checks, and so on;
- The definition of new design concepts to cope with the evolution of programming languages, simplifying the mapping between the design and implementation representations;

- The definition of new design concepts to cope with the evolution of requirements/domain analysis artifacts, simplifying the mapping between the requirement elicitation and design representations.

The work in this dissertation can be classified in the first category, since it extends the basic UML design language with features that are specific to framework development. However, several research efforts in the field have inspired our solution. The following subsections describe two of them: ADVs [3, 20] and the Existence Dependency relationship [92].

2.1.1 ABSTRACT DATA VIEWS (ADVS)

Abstract Data Views [20] and their representation through the views-a relationship [3] help to clarify design diagrams that require object notification mechanisms. The views-a relationship can be seen as a design level representation of the MVC framework [52] state-consistency and notification mechanisms.

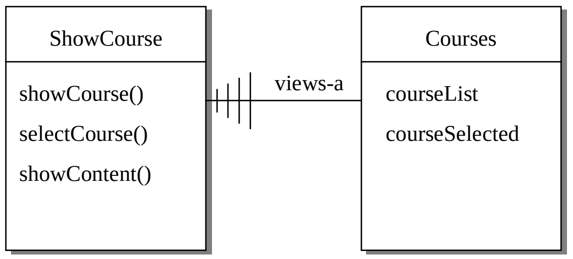


Figure 2.1. The views-a relationship

Figure 2.1 is an example of the use of the views-a relationship: the *ShowCourse* class is a "view" of the *Courses* class. This means that every time the content of the selected course changes (modeled by an instance of the *Courses* class) the object that displays the course information (modeled by an instance of *ShowCourse)* is automatically updated[1].

Once the semantics of the views-a relationship is well understood and formally defined, its use leads to a clearer design without the complication of showing the various notification mechanisms. This notation can simplify the appearance of the design in comparison to the use of design patterns for the same purpose, such as the observer [34].

[1] This example is taken from the ALADIN web-based education framework [31]. ALADIN allows the creation of web-based authoring tools such as AulaNet [21].

The views-a relationship extends the standard OOADM design languages, which are normally based on generalization/specialization, association, and aggregation relationships only. The introduction of this new kind of relationship can largely simplify design diagrams and promote loose coupling as shown in [3, 20].

2.1.2 EXISTENCE DEPENDENCY

Snoeck and Dedene [92] propose a new kind of relationship, namely the existence dependency relationship, and show how every object-oriented system may have its design represented by two kinds of relationship only: existence dependency and generalization/specialization. Basically they show how this new relationship overcomes the necessity of different kinds of association and aggregation relationships.

The great contribution of their work is that the semantics of existence dependency is simple, formally defined (as opposed to the various kinds of associations and aggregations), and can be easily mapped to constructs that already exist in most OO design and implementation languages. They propose that designers should first build models based on existence dependency and generalization/specialization only, achieving designs with precise and well-defined semantics, which can be more easily validated. At a later stage in the design process, or in the implementation process, the existence dependency relationships can be mapped into existing design/implementation elements that preserve its semantics.

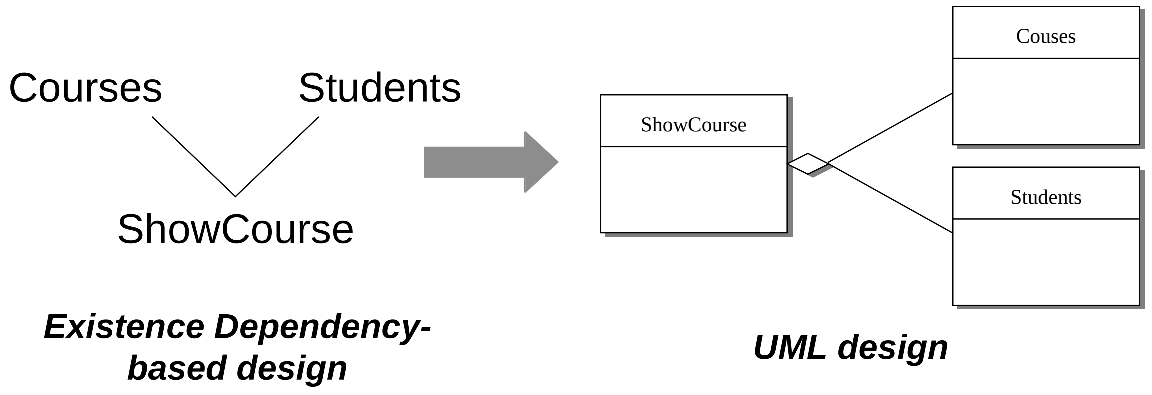


Figure 2.2. Existence Dependency-based design and its representation in UML

Figure 2.2 illustrates how the existence dependency relationship may be used to model the *ShowCourse* semantics: a *ShowCourse* object can exist only if there is a student that wants to browse a course content. Once this relationship is precisely represented by the existence dependency relationship it may be modeled in standard design languages, such as UML.

The use of existence dependency clarifies the design diagrams, since it is a simple and well-defined relationship, and allows for semantic checks at design level as shown in [92].

2.1.3 CONCLUSIONS

Both the works in ADVs [3, 20] and Existence Dependency [92] extend the standard design languages with new constructs that have well-defined semantics and that enhance the representation of object-oriented systems. The solution proposed in this work follows the same general principles, extending the UML design language with new constructs that facilitate the representation of frameworks. The semantics of the new constructs is precisely defined to allow for the construction of tools that automate some framework development activities.

2.2 FRAMEWORKS AND DESIGN PATTERNS

There is always great confusion about whether frameworks and design patterns differ from each other [58]. There are several reasons for this, including the facts that there are no formal definitions for both frameworks and design patterns and that most of the well-known design patterns [13, 34, 95] are related to providing flexibility and extensibility mechanisms to systems, which is a key concept associated with frameworks.

For the purposes of this work a framework is defined as a system that is composed of a kernel subsystem, which is common to all the applications that may be generated within the framework, and a hot-spot subsystem, which implements the application-specific behavior. An application developer generates a framework instance customizing the hot-spot subsystem during the instantiation process.

Design patterns are design techniques that do solve several design problems [29, 34, 61]. However, the discovery, documentation, and classification of new patterns cannot be considered as research in object-oriented design, since these tasks do not require the creation of new design artifacts, processes, or tools, but instead they use the current object-oriented design technology to express solutions for recurring design problems.

The new hot-spot constructs provided by the framework design language model flexibility and extensibility mechanisms as built-in elements, overcoming the need to use design patterns for these purposes at design level. However, design patterns are considered as a possible approach for mapping these new design constructs into appropriate constructs of current OO languages.

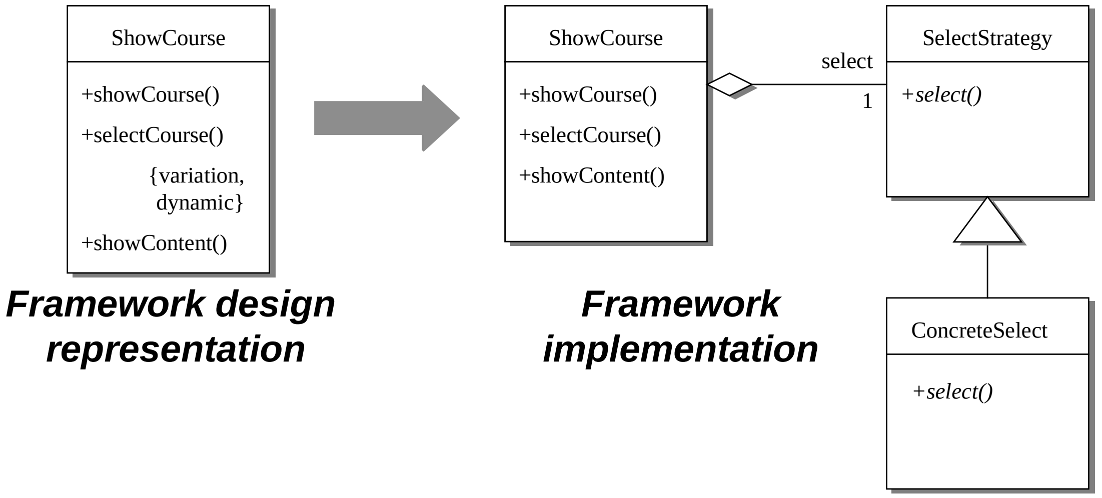


Figure 2.3. Implementing hot-spots with design patterns

Figure 2.3 shows the representation of a hot-spot in the proposed UML-extended design language, where the tagged values *{variation, dynamic}* apply to method *selectCourse()* indicating that its implementation can vary at run-time and depends on the framework instantiation. A possible implementation solution to this flexibility problem is the strategy design pattern [34], which allows multiple algorithms to be used depending on a given context. Figure 2.3 shows how a mapping from the framework design representation to standard UML can be made, using the strategy design pattern to "implement" the *selectCourse()* hot-spot.

As will be shown in chapter 6, there are several other possible approaches to realize this mapping, such as meta-level programming [51], aspect-oriented programming [50], and subject-oriented programming [38].

Architectural patterns [13] and architectural styles [1, 87] may be used to assist framework development [13]. However, they will not be considered in this dissertation, and their relationship with the proposed framework design language is a hot topic for possible future investigations.

2.3 TRANSFORMATIONAL SYSTEMS AND KBSE

Transformational systems are tools that give support to the transformations of software artifacts. They may be classified into two categories: specific and generic. Specific transformational systems are the ones that give support to a limited number of artifact description languages and operators, while generic transformational systems are open architectures that may be configured for different purposes [83]. Compilers can be classified as specific transformational systems, while Draco [64, 84] and TXL [18] are examples of generic transformational systems.

The transformations used in software development tasks may be classified as vertical or horizontal transformations. Vertical transformations convert artifacts from one level of abstraction to another, which can be either more abstract or more concrete. Transformations that generate code from design (and vice-versa) are examples of vertical transformations. Horizontal transformations preserve the level of abstraction of the artifacts involved. A common example of horizontal transformations is the conversion of programs written in one programming language to another (e. g. COBOL to C++ [54]).

The tools described in this dissertation are examples of specific transformational systems that support both, vertical and horizontal transformations. The transformation illustrated in Figure 2.3 is a vertical transformation, since it maps a design representation (more abstract) into an implementation representation (more concrete). Figure 2.4 shows an example of horizontal transformation, which represents a refactoring procedure [47, 66] that moves common behavior to abstract super classes. In this example, since the same method *selectCourse()* belongs to classes *ShowCourse* and *Registration* it can be moved into an abstract common super class, *SelectCourse*, without changing the system behavior.

This transformation is a horizontal transformation, since it does not alter the level of abstraction of the artifacts involved.

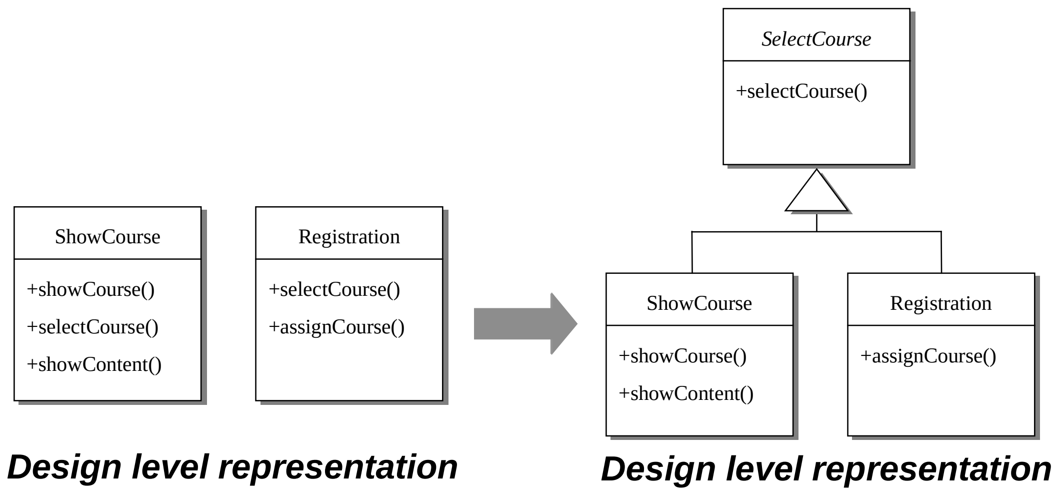


Figure 2.4. Horizontal transformation

Knowledge-based software engineering (KBSE) tools collect and represent the behavior of systems into knowledge bases (KB). KBSE tools have been used to support a variety of software engineering activities such as program understanding [48, 55] and debugging [86], support to business processes [44], and software maintenance [97]. The use of knowledge base as a means for guiding the application of transformations has also been adopted by several transformational environments [84, 90, 91].

```
applyStrategy(Project, NewProject) :-
          [...]
          forall(variationMethod(Project, Class, Method, dynamic),
          strategy(Project, NewProject, Class, Method)),
          [...]

strategy(Project, NewProject, Class, Method) :-
          concat(Method, 'Strategy', NewClass),
          createClass(NewProject, NewClass, dynamic),
          createMethod(NewProject, NewClass, Method, public, none, abstract),
          createAggregation(NewProject, Class, NewClass, strategy),
          assert(implementation(Project, [...], strategy)).
          [...]
```

Searches for variation hot-spots in a design

Uses strategy pattern to model them

Figure 2.5. Transformation sample code

All the tools described in this work have been implemented in PROLOG. They use the PROLOG database to store the software artifacts. Transformational rules are responsible for altering the database as artifacts are created or modified. The transformation illustrated in Figure 2.3 can be described as a logic program that manipulates a knowledge base as described in Figure 2.5.[2]

The *applyStrategy* predicate searches for all the hot-spots marked as *{variation, dynamic}* in a project and applies the strategy design pattern to them. The *strategy* predicate is the one responsible for transforming the design. The history of the transformations applied to a project is also stored in the PROLOG database. In this example, the information that the strategy design pattern was used to implement *{variation, dynamic}* hot-spots is provided by the assertion of the *implementation* predicate into the system's knowledge base.

2.4 FORMAL METHODS

Several approaches have been proposed for the formalization of object-oriented design techniques [3, 4, 9, 25, 26, 28, 40, 92, 94]. These efforts serve as a basis for rigorous software development, crucial for some application domains like real-time systems [53].

For the purposes of this work, formalization is used to provide an unambiguous description of the proposed framework design language and allow the construction of supporting tools. The formalization is based on the description of the semantics of the language through the use of set theory and meaning functions.

Figure 2.6 illustrates the approach. The framework design descriptions are represented in the syntactic plan while their semantics is represented in the semantic plan. The function *Meaning(design)* is responsible for mapping each design description to its semantic representation, which is the set of applications that may be instantiated from it.

Through this formalization, a framework can be defined as any system in which *#Meaning(design) > 1*, where *#Set* is the cardinality of the set *Set*. This means that a framework is any system that allows the instantiation of more than one application. For every application that is not a framework *#Meaning(design) = 1*. Figure 2.6 shows two design representations: the first, *design*, is a framework since there are several

[2] The application of the transformation described in Figure 2.5 would generate a class called *SelectCourseStrategy* which would contain a method called *selectCourse()*. Figure 2.3 uses shorter names to simplify the diagram: *SelectStrategy* to represent the class name and *select()* to represent the method.

applications that can be instantiated from it; the second, *otherDesign,* is not a framework and *#Meaning(otherDesign) = 1.*

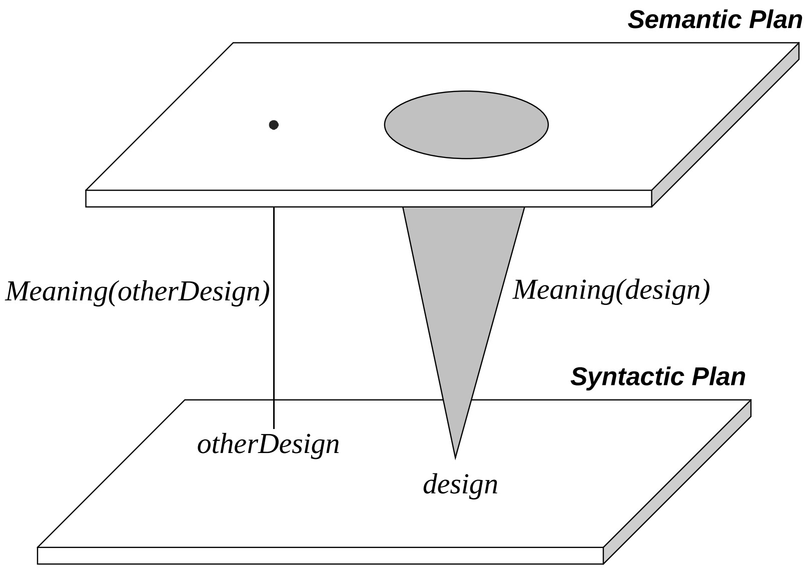


Figure 2.6. Formalization approach

The verification that a given application is a valid instance of a given framework can be made by the *IsInstance* function. If *application* and *framework* are the design descriptions of the application and framework, respectively, then *IsInstance(framework, application)* returns TRUE if *Meaning(application)* $\subset$ *Meaning(framework)* and FALSE otherwise (Figure 2.7).

A transformation is modeled as a mapping between two design descriptions in the syntactic plan. In the case of behavior-preserving transformations, as the ones illustrated in Figure 2.3 and Figure 2.4, the equation *Meaning(design) = Meaning(Transformation(t, design)),* needs to hold. *Transformation(t, design)* returns the syntactic description of the transformed design, after the application of *t*. Figure 2.8 shows a geometric view of a behavior-preserving transformation.

Chapter 9 describes how the function *Meaning* is defined and how the transformations presented throughout this dissertation may be validated. A study of the computational complexity of the algorithms used to calculate *Meaning, IsInstance,* and verify the behavior-preserving transformations is also presented to show that tools supporting these verifications may be constructed. *Verifier* is described as an example of a tool that implements these semantic checks and can be used as a basis for a rigorous framework development process.

A consequence of the formal description of the design language semantics is the precise definition of what frameworks really are, which is still an open issue [46].

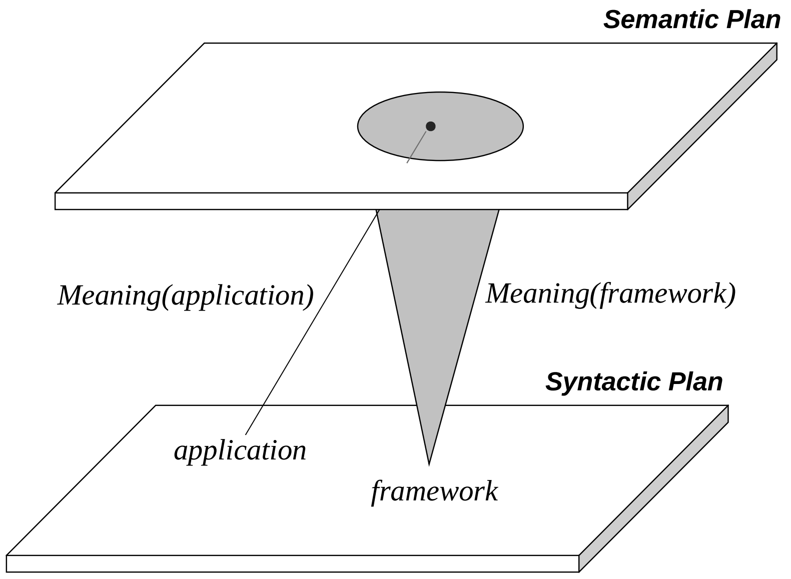


Figure 2.7. Frameworks and framework instances

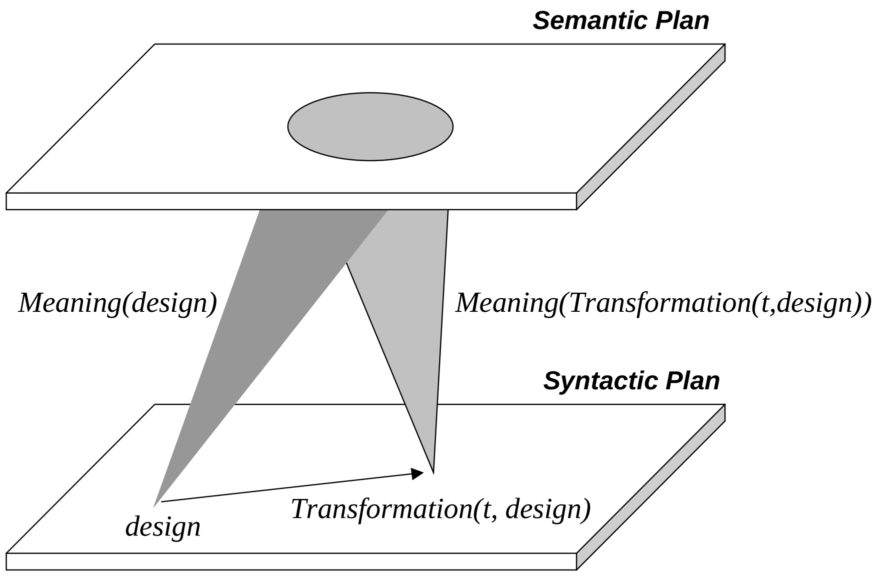


Figure 2.8. Behavior-preserving transformation

2.5 SUMMARY

The research in object-oriented design focuses on developing new design constructs to enhance the representation of systems at higher levels of abstractions than programming languages. The work on ADVs [3, 20] and Existence Dependency [92] are good examples: both extend object-oriented design languages with useful and formally defined concepts.

Based on these ideas, this dissertation proposes an extension to the UML design language to enhance the representation of object-oriented frameworks. Since some of the new constructs added to UML do not have a direct mapping into existing OO programming languages, modeling techniques have to be used to support their implementation. The design patterns catalogs [13, 34, 95] list several patterns that can be used to support this mapping, however, there are other possible solutions including meta-level programming [51] and subject-oriented programming [38].

The mapping from the new design constructs to existing implementation level constructs can be seen as a vertical transformation, since a more abstract design artifact is transformed into a more concrete implementation artifact. Refactorings [47, 66], on the other hand, represent horizontal transformations, since they preserve the level of abstraction of the involved artifacts.

The tools described in this dissertation are transformational systems that use a knowledge base to guide the application of the transformations as well as to represent the software artifacts. Current implementation of these tools is based on PROLOG.

The formalization approach used to precisely describe the semantics of the framework design language is based on set theory. The semantics of each design description is the set of applications that can be instantiated from it. This formalization allows for the construction of new supporting tools. The verification of the transformations described throughout the dissertation can be done based on the proposed semantics.

Chapter 3 – Framework Design Language

This chapter presents several object-oriented design techniques and discusses their ability to represent frameworks adequately. The framework design language, which is the central part of this work, is then presented as an extension to UML. The new constructs are defined using the UML extensibility mechanisms: stereotypes, tagged values, and constraints. Finally, a brief analysis of the language is presented.

3.1 CURRENT APPROACHES TO FRAMEWORK DESIGN

This section describes some of the current design techniques used to model frameworks, showing that they do not provide constructs to represent the frameworks' hot-spots adequately.

3.1.1 STANDARD OOADMS

Standard OO design methods, like OMT [81] and UML [82], provide static (class and object diagrams) and dynamic (activity, interaction, state-chart, and state diagrams) views of a system design. Figure 3.1 is a representation of a student subsystem of the ALADIN web-based education framework [31] in UML, where (a) shows a static view of the system (class diagram) and (b) shows a dynamic view (interaction diagram). The dynamic view shows the interaction between an instance of each of the two classes.

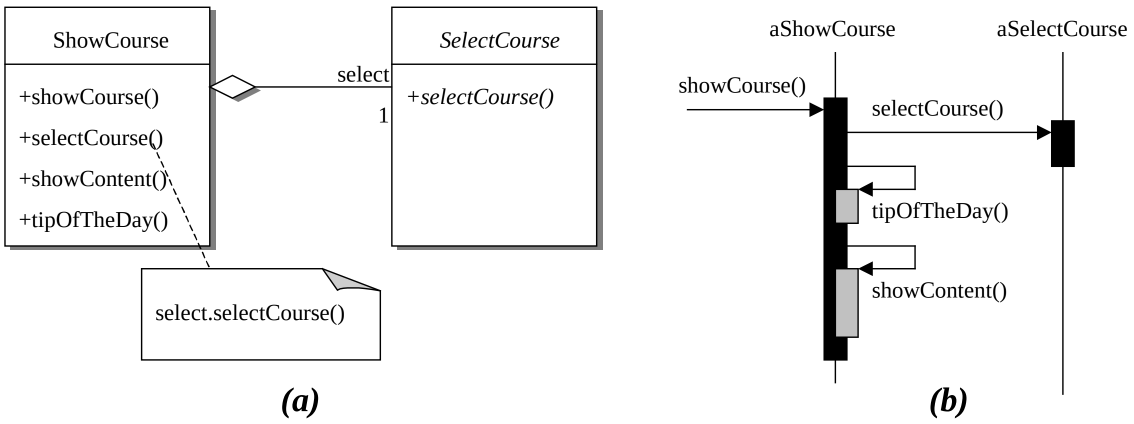


Figure 3.1. UML representation of a framework

The *showCourse()* method is the one responsible for controlling the application flow: it calls *selectCourse()*, which allows the student to select the desired course, *tipOfTheDay()*, which shows a start-up tip, and finally *showContent()* to present the content of the selected course.

The method *selectCourse()* is the one responsible for allowing the student to select the course he or she wants to attend. It is a framework hot-spot since it can have different implementations in different web-based applications created based on the framework (framework instances). Different examples of common course selection mechanisms include: to require a student login or not, to show the entire list of available courses or just the ones related to the student major field, to show a course preview or not, and so on. There are an infinite number of possibilities that depend on the application specific requirements.

Figure 3.1 shows *selectCourse()* as an abstract method of an abstract class, *SelectCourse*. The framework user has to create a *SelectCourse* subclass and provide a concrete implementation for the *selectCourse()* method to instantiate this hot-spot. The problem with this representation is that there is no indication that this method is a hot-spot in the design diagrams. There is also no indication of how the hot-spot should be instantiated. Although the name of the abstract method *selectCourse()* is italicized, this notation is not an indication of a hot-spot, rather it indicates an abstract method. Abstract methods do not necessarily represent framework hot-spots.

The method *tipOfTheDay()* is also a framework hot-spot. The reason is that it may be wanted by some instances but unwanted by others. The framework should provide in its kernel classes only the methods and information useful to all possible instantiated applications. The extra methods, which may be referred to as instance methods, should only be added to the framework instances. The inclusion of instance methods to framework classes could result in a complex interface for those classes, which is a bad design heuristic [76]. Changing the implementation of an existing class every time new functionality is needed is also not good practice. A notation to designate where instance methods should be added could avoid this problem and would be extremely useful in framework design.

This simple example shows that standard OO modeling notations could be usefully augmented with constructs to distinguish hot-spots from kernel elements.

Different OO design notations include different artifacts, such as the representation of object responsibilities as CRC cards [8, 98]. However none of these artifacts are helpful for explicitly representing framework hot-spots and how they should be instantiated. Other models, such as data-flow and entity-relationship diagrams, could also benefit from an approach for annotating hot-spots.

3.1.2 CONSIDERATIONS ABOUT DESIGN PATTERNS AND META-LEVEL PROGRAMMING

Design patterns [13, 34, 95] are usually described using standard OO notations such as OMT and UML. However, since various design patterns provide solutions to variability and extensibility problems [34] they define a common vocabulary to talk about these concepts [95] and may enhance the understanding of framework designs.

Figure 3.2 is an example of the use of design patterns to model the hot-spots of a framework. It uses the strategy design pattern [34] to model the *selectCourse()* hot-spot and the visitor design pattern [34] to model the *tipOfTheDay()* hot-spot. Note that the use of visitor avoids the maintenance problem of *ShowCourse*, since it allows the extension of the class interface without changing it directly.

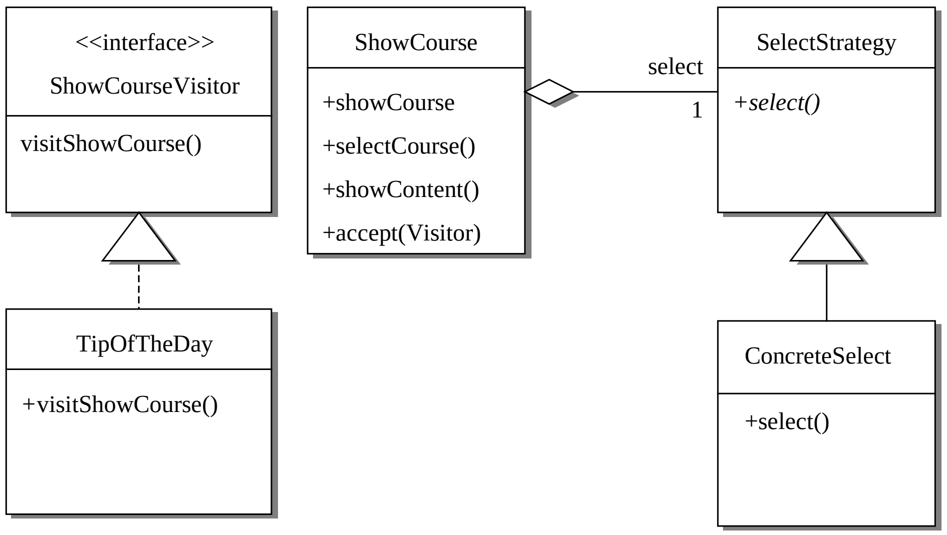


Figure 3.2. Using design patterns to model frameworks

In this example the design pattern names are used in the class names. This approach can help the framework user in finding the framework hot-spots, but only when he or she knows the design patterns that were used for modeling each of the hot-spots. In a typical framework design a single hot-spot class can be used (with different roles) in various design patterns. Then the approach of using design pattern names as class names fails[3]. In addition, representing design patterns with standard OO design notations can lead to very complex designs, even for simple frameworks (this point will be discussed in more detail in chapter 10, when case studies are presented).

Meta-level programming [51], which can be seen as an architectural pattern [13], provides a good design solution for allowing run-time system reconfiguration. Meta-level programming belongs to the same category of design patterns: it does not provide extensions to design languages and is generally represented using standard design techniques. However, if the hot-spots are expressed through meta-object protocols (MOPs) they can be more easily identified in the design model.

Figure 3.3 shows an example of the use of meta-programming to model the *selectCourse()* hot-spot. Class *SelectMOP* defines the MOP for the definition of the course selection procedure. This class stores the current procedure and allows its modification through the *editSelectCourse()* method, which uses the classes *Courses* and *Students* to configure the possible selection mechanisms. In this case, the course selection algorithm is said to be a meta-level concept while courses and students are base-level concepts. Meta-level concepts are always defined in terms of base-level ones. The course selection procedure is internally represented in the *SelectMOP* class. When method *selectClass()* is called the algorithm defined by the last invocation of *editSelect()* is executed.

The use of meta-level programming is a good alternative for modeling hot-spots that require run-time instantiation (or dynamic hot-spots). On the other hand, hot-spots that need to be instantiated at compile-time cannot be modeled by meta-programming. An advantage of the meta-programming solution, when compared to the strategy design pattern solution, is that it is more restrictive (or black-box) since the only possible method definitions are the ones allowed by the MOP. In the strategy approach any

[3] This is not a problem of the design patterns themselves, but of how they are represented at design level. The role-based modeling technique [75], for example, can be used to solve this problem as described in section 3.1.3.

method can be defined since the instantiation is made through class inheritance and method overriding.

Both design patterns and meta-level programming can help with the identification of the frameworks hot-spots, but they still do not solve the problem (and are not supposed to, since they are not design languages but solutions to common design problems). Also, they are only helpful if the framework user knows the design patterns that were used and how the MOPs were structured.

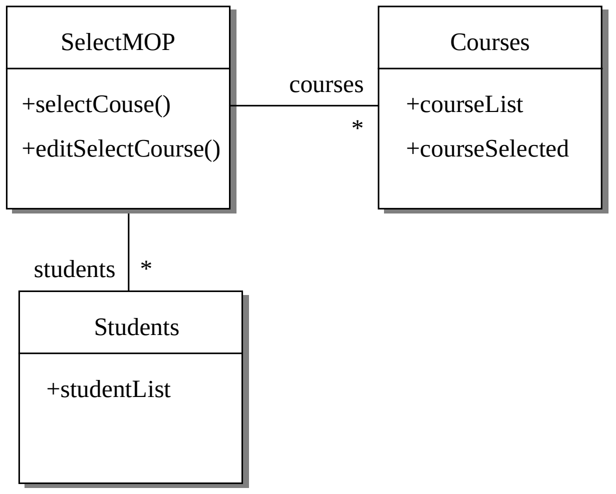


Figure 3.3. Meta-programming example

3.1.3 OBJECT-ORIENTED MODEL WITH ROLES

The use of role diagrams to represent object collaboration is one of the most promising fields in object-oriented design research [17]. Riehle proposes an extension of the OOram methodology [74] to facilitate framework design and documentation [75]. His work proposes a solution for an explicit division of the design, highlighting the interaction of the framework with its clients.

Figure 3.4 is an example of the use of roles in framework design. *ShowCourse* plays the role of a client from the point of view of the strategy design pattern [34], choosing an appropriate course selection mechanism that is then implemented by subclasses of *SelectCourse*. *ShowCourse* is also an observer [34] of the *Courses* class, assuring that the content of the currently selected course will be the one presented.

The use of roles does simplify the modeling of patterns that require a lot of object collaboration and provides a solution for documenting classes that participate in several design patterns at the same time. However, no distinction is made between the kernel and hot-spot elements. This problem is handled using design patterns in the same way as proposed in section 3.1.2. If the framework user knows what patterns were used to model each of the hot-spots he or she can have an intuition on how the framework should be instantiated. On the other hand, if the pattern selections are not explicitly represented, the identification of the framework hot-spots becomes very hard.

Another disadvantage of this approach is the solution for modeling unforeseen extensions of framework proposed in [75], which is in fact a design pattern [7], may lead to a very tangled design. Although it can be a good solution it should have a more concise representation at the design level.

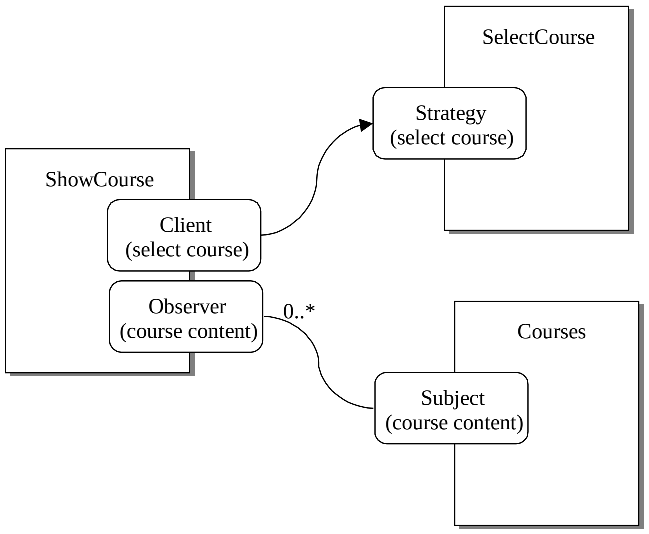


Figure 3.4. A role-based modeling example

3.1.4 CATALYSIS AND THE UML BINDING STEREOTYPE

UML represents design patterns as collaborations (or mechanisms) and provides a way of instantiating framework descriptions through the binding stereotype [82]. Figure 3.5 shows how the *selectCourse()* hot-spot may be represented in UML, where *Course Selection* represents the framework subsystem that models this hot-spot, *SelectCourse* represents the role that has to be instantiated, and *SimpleSelection* models a possible course selection mechanism provided during the framework instantiation. The dashed relationship between *Course Selection* and *SimpleSelection* is the UML binding stereotype, which is used to model the instantiation of patterns and frameworks. In this example it means that the *SelectCourse* role has been instantiated by the *SimpleSelection* concrete class.

However, framework instantiation can be far more complex than simply assigning concrete classes to roles: new classes may have to be created, abstract methods may have to be implemented, and so on. Catalysis follows the UML approach and proposes a design method for building frameworks [23].

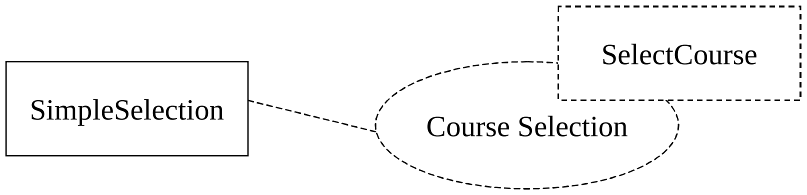


Figure 3.5. Representing collaborations in UML

3.1.5 HOT-SPOT CARDS AND META-PATTERNS

Pree recognizes that standard OO design methods neglect the identification of hot-spots, and proposes the use of hot-spot cards as a means for documenting the information required for developing frameworks [70, 71]. The cards document the semantics of each framework hot-spot, and are divided into two categories: data and function hot-spot cards. Since hot-spot cards are meant to cover the gap between the framework developers and domain experts (who are generally not software engineers), they avoid terms such as classes and objects.

Course selection criteria *(Function)* **Specify degree of flexibility:** [✓ adaptation without restart] adaptation by end user	**Course selection criteria *(Data)*** **Specify the importance:** principal domain abstraction [✓ subordinate abstraction]
(general description of semantics) There can be many possible course selection criteria: courses may or may not require a student login (…).	***(general description of semantics)*** It exists only for allowing several selection criteria (…)
(hot spot behavior in typical situations) It presents a list of all available courses (…)	***(typical instantiation situations)*** select

Figure 3.6. Using hot-spot cards to model the select course hot-spot

Figure 3.6 presents the data and function cards for the *selectCourse()* hot-spot. The degree of flexibility field in the function card is used to model the run-time requirements.

If adaptation without restart is needed then the hot-spot is dynamic, and if adaptation by end user is required it is necessary to build an extra tool to assist with the instantiation process. The *process-based instantiation tool,* described in chapter 7, is an example of a tool that may be used for this purpose. The remaining fields serve as informal documentation.

Hot-spot cards are primarily concerned with capturing domain knowledge to assist with framework development, and are used in the domain analysis/requirements phase. However, Pree proposes the use of meta-patterns, also referred to as design pattern essentials, as a means for documenting hot-spots during design [70, 71]. Meta-patterns describe mechanisms for assembling domain-oriented design patterns. However, like design patterns, they provide flexibility and extensibility solutions based on current object-oriented design constructs and do not really extend object-oriented design languages. Meta-patterns are described using standard OOADM design notations, and consequently, this approach suffers from the same problems as the ones described in the previous sections: hot-spots are not explicitly identified, unless the framework user knows which meta-patterns have been applied to model each hot-spot.

3.1.6 OTHER APPROACHES

The hook tool [33] uses an extended version of UML to model frameworks in which the hot-spot classes are represented in gray. Although this differentiation between kernel and hot-spot classes may help framework design and instantiation, it does not solve the problem completely.

The framework designers still have to provide the solutions for modeling each hot-spot without any support. If instead of basing itself on standard OO constructors, new elements to model the hot-spot semantics were provided by the design language, this task could be simplified and partially automated.

Another extension to standard OOADMs that explicitly represents hot-spots is described in [89]. However, as in the hook approach, the language extensions proposed there are based on existing OO constructs and the hot-spot semantics are not represented as first-class citizens.

3.2 THE FRAMEWORK DESIGN LANGUAGE

This section outlines some requirements for an adequate framework design language. The UML-extended language is then introduced through an example followed by a general description of its syntax and informal semantics. Finally, a discussion of how the proposed language satisfies the requirements is presented.

3.2.1 REQUIREMENTS FOR A DESIGN LANGUAGE

Based on the study of current solutions for framework design and on our experience in modeling and implementing frameworks [30, 31, 77] some requirements for a design language have been derived:

R.1. To provide static and dynamic views of the system. Just like standard OOADM languages, a framework design language should provide a functional view of the framework, representing structural and dynamic aspects of its design;

R.2. To represent the framework's hot-spots so as to differentiate them from the framework kernel. This notation will assist the framework user in identifying more easily the parts of the framework that need to be changed when creating applications, in the instantiation phase;

R.3. To classify and represent the hot-spots with respect to their type. This classification should identify the hot-spot semantics and serve as a basis for the selection of the more appropriate modeling technique for each hot-spot type, in the implementation phase;

R.4. To represent semantic instantiation restrictions. It may be useful to specify semantic restrictions that limit the set of possible instantiated applications. For example, a hot-spot for calculating final grades in a education framework should only return numbers between 0 and 4;

R.5. To represent instantiation process. Since the framework instantiation process can be quite complex it should be explicitly represented to facilitate the job of the application developer, and perhaps allow for the construction of supporting tools.

3.2.2 MODELING EXAMPLE

In the ALADIN web-based education framework [31] the class *Actors* is used to let new types of actors be defined depending on the requirements of a given framework instance. The default actor types are students, teachers, and administrators, however, new types may be needed such as markers, librarians, and secretaries. This design structure is presented in Figure 3.7.

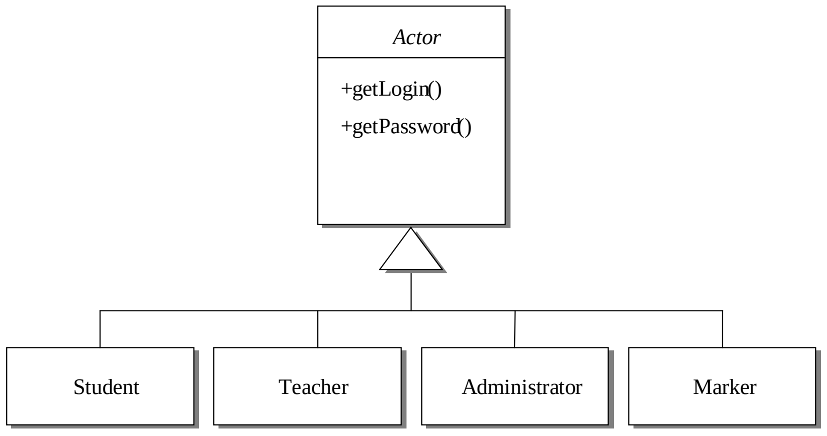


Figure 3.7. Actor hierarchy

This subsection illustrates how the student subsystem and the actor class hierarchy of the ALADIN framework [31] may be modeled using the design language. Figure 3.8 is a representation of this part of the framework with extensions to the UML language to represent and classify the framework hot-spots explicitly.

UML provides three language extension mechanisms: stereotypes, tagged values, and constraints. Stereotypes are extensions to the UML vocabulary. They allow the definition of new building blocks that may be used in conjunction with the standard ones. Tagged values are used to extend the properties of a building block. Finally, constraints may be used to modify the semantics of any UML element.

The tagged values *{variation}* and *{extensible}* extend the UML class definitions to identify two different kinds of hot-spots: variation methods and extension classes, respectively. In this example *{variation}* refers to the *selectCourse()* method, indicating that its implementation varies depending on the framework instantiation; *{extensible}* applies to the *ShowCourse* class, indicating that its interface may be extended during the framework instantiation by the addition of new methods, such as *tipOfTheDay()*.

The *incomplete* constraint is used to identify that class *Actor* is an extension interface hot-spot, meaning that new subclasses of *Actor* may be defined by framework instances. The stereotype *instance class* indicates classes that are not part of the framework structure and belong to instance applications only. The grayed class *NewRole* is an example of an instance class. Incomplete allows several instance classes to be created from a given extension interface hot-spot during the framework instantiation.

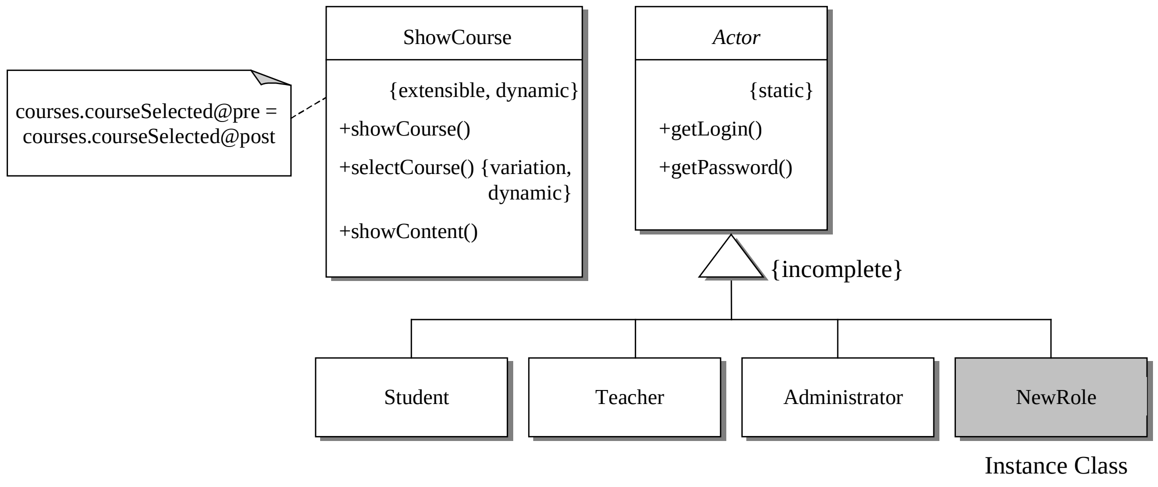


Figure 3.8. Extended class diagrams

Incomplete is a constraint provided by the standard set of UML constraints, which specifies that not all children of the generalization relationship have been defined yet. In this work, its meaning was slightly altered to also encompass realization relationships and to assure that the missing subclasses are instance classes that will be completed only during instantiation.

In this example, instance applications always have three kinds of actors, students, teachers, and administrators, but several other roles may be defined depending on the application specific requirements.

The tagged value *{dynamic}* complements the hot-spot definition by informing that it requires a run-time reconfiguration. Each hot-spot must be identified either by the *{dynamic}* or by the *{static}* tagged value, where this last one is used to indicate a hot-spot that does not need run-time reconfiguration. Variation methods are reconfigured by changing the method implementation. Extension classes are reconfigured by allowing new methods to be defined. Extension interfaces are reconfigured by the creation of new instance classes.

Note that this design is more abstract than the class diagrams represented in plain UML, since the new constructs cannot be directly mapped into constructs available in existing OO programming languages. The class diagrams in standard UML are low-level diagrams that resemble the structure of current OO languages, like Java and Smalltalk. The new diagrams are also more concise. In this example the class *SelectCourse* presented in Figure 3.1 is not required, simplifying the design model.

The design language provides two ways for specifying semantic restrictions on the hot-spots: using notes and template diagrams.

The note attached to the *SelectCourse* extension class is an OCL [65, 82] specification that defines that the class attribute *courseSelected* cannot be changed by any of the new methods that may be added during framework instantiation.

Template diagrams are an alternative way of limiting hot-spot behavior. An example is presented in Figure 3.9(a), where the *{optional}* tagged value indicates interactions that may not occur, and depend upon the framework instantiation *(loginPage* and *validateData*). Figure 3.9(a) tells us that a concrete method that instantiates the *selectCourse()* variation method hot-spot must have the following behavior:

1. It *may* display a login Web page;
2. It *has to* show a Web page for the selection of the desired course. This page may display only the courses to which the student is assigned (if step 1 is performed) or it may show all the courses. In addition, the student information may be provided in this page (if step 1 is not performed). As illustrated in Figure 3.9(b), this step is also a hot-spot. This point is discussed next;
3. It *may* validate the data by checking if the login is valid, and whether the student is assigned to the course or not. This step is optional since there can be courses that do not require student identification;
4. It *has to* set the value of the selected course as the one being presented.

As described in item 2, hot-spots can be composed of other hot-spots. Figure 3.9(b) describes instantiation restriction that applies to the *selectionPage()* hot-spot: it cannot use more than HTML 3 frames.

In order to keep template diagrams simple, more than one diagram may be provided for each hot-spot. In this case, one of them has to be followed by the instance methods defined during instantiation. If no template diagram or OCL specification is provided the hot-spot can be instantiated by any method that follows its signature. The template diagrams and the OCL specifications are complementary since they both restrict the behavior of the hot-spots.

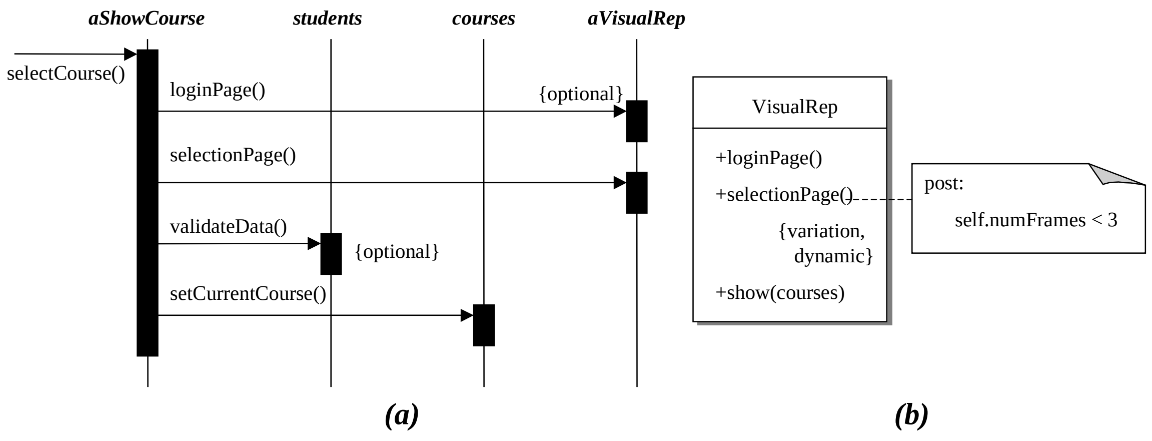


Figure 3.9. Template diagram (a) and OCL specification (b)

Figure 3.10 shows the instantiation diagram for the framework illustrated in the example. It is a representation of the instantiation process that uses the same syntax of the UML activity diagrams. Activity diagrams are used to represent workflow procedures in standard UML [82].

Several facts related to the framework instantiation may be derived from this diagram. The extension of *ShowCourse* interface is optional, since it may not be required by a given framework instance. On the other hand, it may be extended with several methods. The same is true for the *Actor* interface extension hot-spot: several *Actor* subclasses may be defined by a given application but none may be needed by another. The *selectCourse()* hot-spot is not optional and has to be configured by every framework instance. Finally, only one course selection algorithm is allowed per instance.

The extended class diagrams, template diagrams, and the instantiation diagrams complement each other, and together completely specify the framework hot-spots, its instantiation restrictions, and how the framework should be instantiated.

It is very important that framework developers provide documentation that describes what parts of the system should be changed to create a valid framework instance since it is very unlikely that a framework user will be able to browse the framework code, which generally has complex and large class hierarchies, and write the appropriate code if the framework is not well documented. These diagrams address this problem.

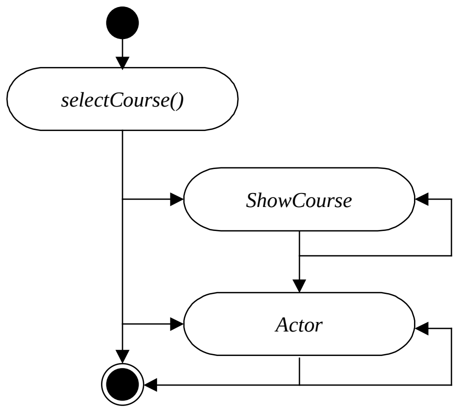


Figure 3.10. Instantiation diagram

3.2.3 LANGUAGE DESCRIPTION

There are three kind of hot-spots in object-oriented frameworks: variation methods, extension classes, and extension interfaces. Variation methods are methods that have a well-defined signature, but whose implementation may vary depending on the system instantiation. In the example, method *selectCourse()*, is a variation method. Extension classes are classes that may have their interfaces extended during the framework instantiation. The *ShowCourse* class, for example, may require the addition of new instance methods (like *TipsOfTheDay())* for each different instantiated application. Extension interfaces are interfaces[4] or abstract classes that allow the creation of concrete subclasses during the framework instantiation. The instantiation of this last kind of hot-spot takes place through the creation of new classes, the instance classes, which exist only in framework instances.

It should be clear that these three kinds of hot-spots have different semantics: in variation method hot-spots the method implementation varies, in extension classes the class interface varies, finally, in extension interfaces the types in the system vary (new instance classes may be provided). All three kinds of hot-spots may either be static (do not need to be reconfigured at run-time) or dynamic (require run-time reconfiguration).

[4] Interfaces are built in concepts in some languages like UML and Java.

These new constructs are represented as extensions to UML by:

- Extending the class diagrams to explicitly identify and classify the framework hot-spots;
- Defining template diagrams to hot-spot model instantiation restrictions;
- Defining instantiation diagrams to represent the framework instantiation process.

UML class diagrams are extended by a new stereotype, *instance class*, and four new tagged values, *variation, extensible, static* and *dynamic*. The first two represent variation methods and extension classes, respectively. *Static* and *dynamic* are used to classify the hot-spots regarding to their run-time requirements. The applicability of the *incomplete* constraint has been modified to identify extension interface hot-spots.

The constraint *restricted* is also part of the framework design language. It limits the behavior of the extensible interfaces by not allowing the addition of new methods to the instance classes. If a given interface is restricted its subclasses can override the interface methods but cannot implement new ones. It is correct to think that subclasses of unrestricted extension interfaces are extensible classes while subclasses of restricted extension interfaces are standard classes. The use of *restricted* will be exemplified several times throughout the dissertation.

The keywords extensible, variation, incomplete, and restricted indicate what are the framework's hot-spots and their exact meaning. The instance class stereotype indicates where in the system new classes may be defined during instantiation. Table 1 summarizes these elements and defines their semantics.

OCL specifications [65, 82] may be written on notes as in standard UML, however, they have a different semantics if the notes are attached to hot-spots. In the case of variation methods, it means that all method implementations that may be defined during instantiation should follow the specification. In the case of extension classes, it applies to all methods that might be added during instantiation. In the case of extension interfaces, it applies to all methods that may be overridden and added to each instance class.

Element	Type	Applies to	Semantics
Instance Class	Stereotype	Class	Classes that exist only in the framework instance. New instance classes are defined during the

			framework instantiation.
Variation	Tagged value	Method	It means that the method implementation depends on the framework instantiation. It is used to identify variation method hot-spots.
Extensible	Tagged value	Class	It means that the class interface depends on the framework instantiation: new methods may be defined to extend the class functionality. This element identifies extension class hot-spots.
Static	Tagged value	Extension Interface, Variation Method, and Extension Class.	It means that the hot-spot does not require run-time reconfiguration.
Dynamic	Tagged value	Extension Interface, Variation Method, and Extension Class.	It means that the hot-spot requires run-time reconfiguration.
Incomplete	Constraint	Generalization and Realization	It has almost the same meaning as in standard UML. Incomplete means that new classes that satisfy a given relationship (generalization or realization) may be added during the framework instantiation, identifying extension interface hot-spots.
Restricted	Constraint	Incomplete Generalization and Incomplete Realization	It means that the instance classes created from the relationship should not extend its interface. This element identifies restricted extension interfaces.

Table 3.1. Summary of the new elements and their meanings

Template diagrams extend the standard interaction diagrams with the new tagged value *optional*, which indicates that an event may or may not occur. Template diagrams may be applied to all hot-spots, just like another kind of OCL specification. For each framework hot-spot a finite set of template diagrams may be provided, in which case at least one of them must be followed by the hot-spot instances. Template diagrams and OCL notes may be used in conjunction for a given hot-spot, but in this case the framework designer should guarantee that the two specifications do not conflict (which is computationally undecidable). Generally, template diagrams are used to describe a "pattern behavior" that should be followed by the hot-spot instances, as shown in Figure 3.9(a). OCL specifications, on the other hand, are generally used to specify invariants that should be satisfied by the hot-spot instances, as shown in Figure 3.8 and Figure 3.9(b).

The instantiation diagrams are a visual representation of the framework instantiation process. They are represented using the syntax defined by the UML activity diagrams. Each action state is used to represent one framework hot-spot. All hot-spots should be modeled in the framework instantiation diagram.

3.2.4 SATISFYING THE REQUIREMENTS

The proposed design language is an extension to UML, which directly meets requirement R1. Requirements R2 and R3 are achieved by the new design elements since they identify and classify each framework hot-spot. The use of OCL [65, 82] and template diagrams to specify behavioral restrictions that limit the possible instantiations addresses R4. Instantiation diagrams explicitly represent the framework instantiation process, as required in R5.

Satisfaction of these requirements is a starting point for showing that the proposed UML-extended design language is adequate for modeling frameworks. The case studies presented in chapter 10 emphasize this point by showing how the language can be applied to model real-world frameworks. Chapters 4 to 8 show that it can be used to help the systematization of several tasks in the framework development process.

3.3 ANALYZING THE LANGUAGE

Standard OOADM languages do not provide constructs for representing flexibility and variability requirements, which have to be represented as a combination of standard OO constructs. The proposed extensions to the UML design language addresses this problem representing hot-spots as first-class citizens and making the framework design more explicit and simple. The new language elements are not concerned with how to implement the variability and extensibility aspects of the framework, but just with how to appropriately represent them at the design level. Consequently, the diagrams are more abstract (and more concise) than standard OOADM diagrams. Unfortunately some of the new design elements cannot be directly mapped into existing OO programming languages.

Extensible interfaces can be directly implemented through standard inheritance. Although dynamic extensible interfaces are not supported in compiled languages such as C++, they may be simulated through dynamic linking (Microsoft Windows DLLs, for example). Variation methods and extension classes, on the other hand, cannot be directly implemented, since standard OO programming languages do not provide appropriate constructs to model them.

To bridge this design-implementation gap, several techniques may be used. Design patterns are a possible solution, since several patterns provide solutions flexibility and extensibility problems and are based only on extension interfaces. Thus, design patterns may be used to transform variation methods and extension classes hot-spots into extension interface hot-spots. Figure 3.11 illustrates the use of the strategy design pattern to implement this mapping[5]. Note that *SelectStrategy* is a restricted extension interface since new methods should not be added to its subclasses.

The transformations used to map variation methods and extensible classes into implementation level constructs must be behavior-preserving, since the system functionality is independent of the implementation technique used to model the hot-spots.

[5] This figure is an enhancement of the transformations illustrated visually by Figure 2.3 and textually by Figure 2.5, since the framework implementation was not appropriately represented there through the use of extension interfaces.

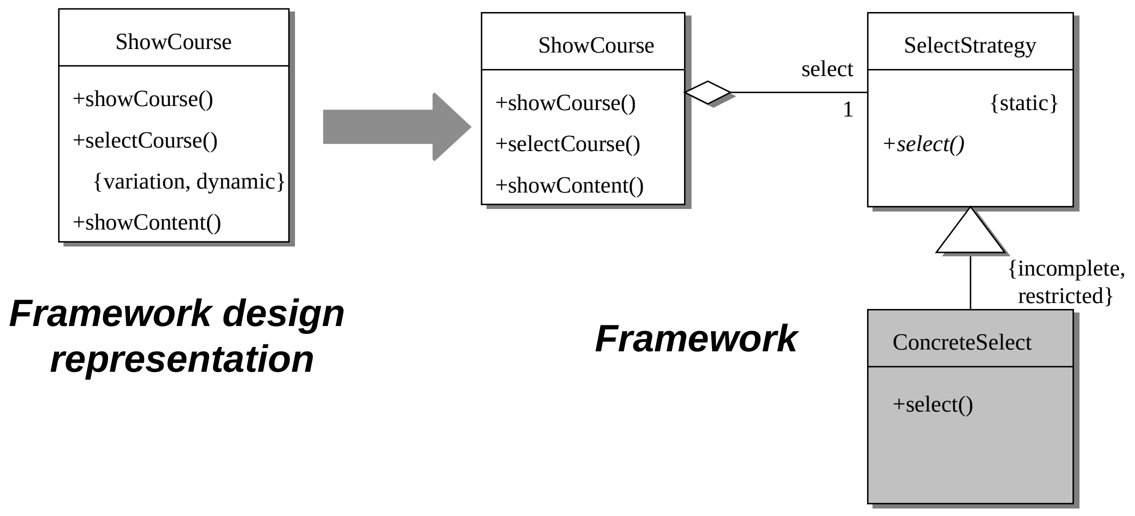


Figure 3.11. Transforming variation methods into extension interface hot-spots

During the framework instantiation, instance classes must be provided to complete the definition of the extensible interface hot-spots (at this point these are the only kind of hot-spots in the system, given that the other two have already been transformed). Figure 3.12 illustrates a framework instantiation.

3.4 SUMMARY

Several approaches have been used to model frameworks, including standard OOADMs [81, 82], meta-patterns [70, 71], and role diagrams [75]. However, none of these approaches explicitly represent and classify framework hot-spots. An extension to the UML design language is proposed to address this problem. This extension is based on the definition of new stereotypes, tagged values, and constraints. OCL notes and template diagrams are used to specify instantiation restrictions on the framework hot-spots. Instantiation diagrams are proposed to represent the instantiation process.

There are three kinds of framework hot-spots: variation methods, extension classes and extension interfaces. The first two cannot be directly represented in standard OO programming languages due to the lack of appropriate constructs. However, several implementation techniques may be used to transform variation methods and extension classes into extension interface hot-spots without changing the semantics of the framework. These transformations occur during the implementation phase and are discussed in more detail in chapter 6. Framework instantiation generally consists of defining instance classes to complete the extension interface hot-spots definitions.

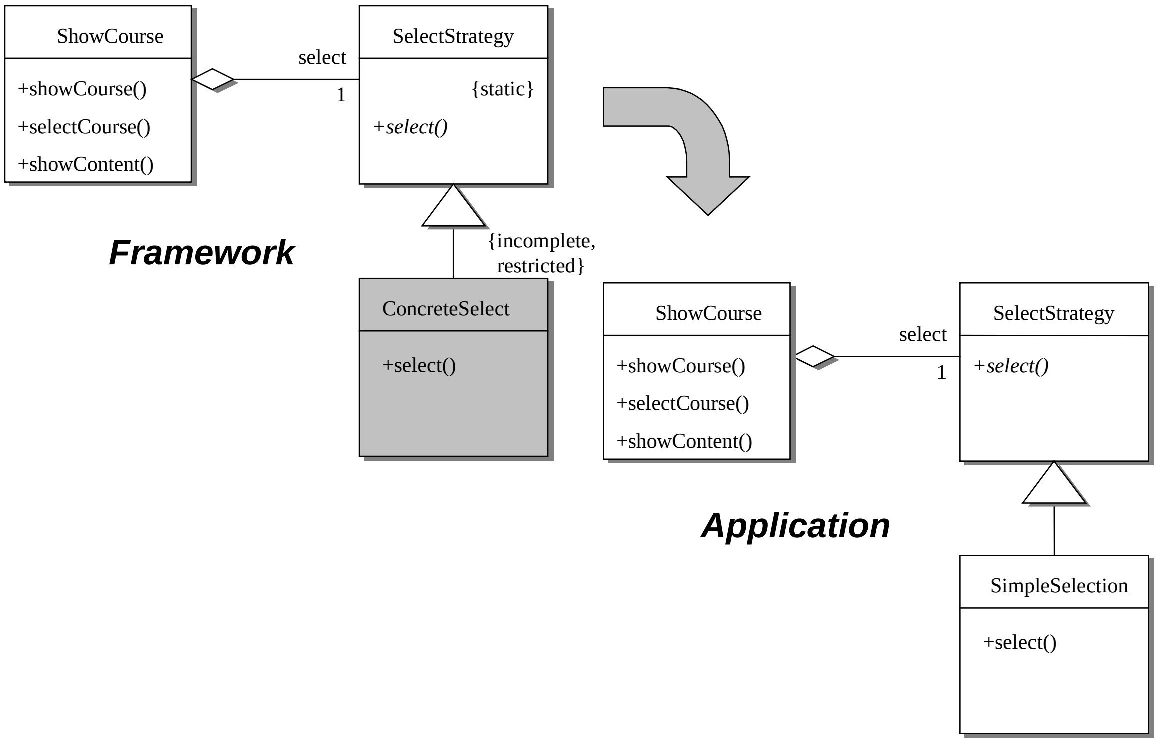


Figure 3.12. Instantiation example

Chapter 4 – Framework Development Process

This chapter presents a transformational approach for the framework development process. Tools used to assist each of the process steps are described at a high level of abstraction. The following chapters (5 through 8) describe each of the tools in detail. The process and the tools described in this dissertation are an example of supporting technology that may be constructed based on the proposed framework design language. Other processes and tools that best fit the needs of specific projects and organizations may be defined.

4.1 A TRANSFORMATIONAL APPROACH TO THE PROCESS

Figure 4.1 illustrates a framework development process, showing the different actor roles, that interact and the artifacts that are consumed and produced in each step.

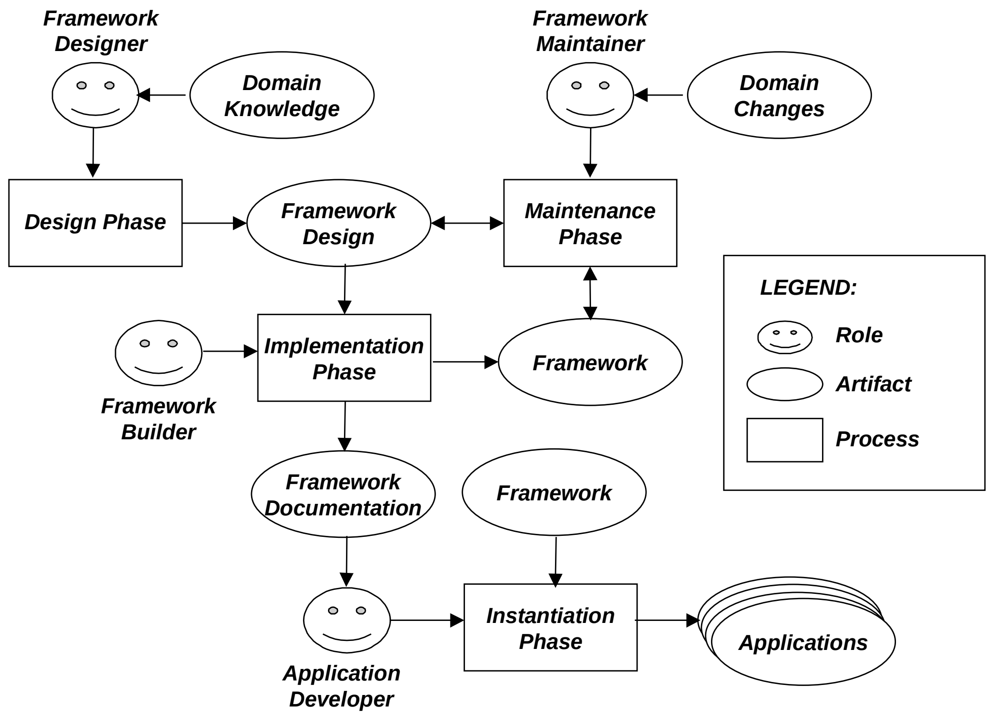


Figure 4.1. Framework development process

Each different development task is assigned to a different role (framework designer, framework builder, framework maintainer, or application developer). However the same actor (and group of actors) can play different roles at the same time in a specific project.

Domain knowledge and domain changes represent artifacts that are produced from the domain analysis/requirements step. Since the tools proposed in this work cannot handle these artifacts they have to be processed by human actors: the framework designer and the framework maintainer, respectively.

Each process is responsible for transforming one kind of artifact: design phase creates the framework design; implementation phase generates the framework and the framework documentation from the framework design; instantiation phase uses the artifacts produced in the implementation phase to generate applications based on the framework. Finally, the maintenance phase can evolve both the framework design and the framework itself, since it supports design and implementation level transformations. All of these transformations may have to be assisted by the actors that are performing the respective roles since information may have to be provided.

The artifact framework appears twice in the figure just to simplify the interconnections.

4.2 THE DESIGN PHASE

In this phase the framework structure is defined by the framework designer, which uses the information generated in the domain analysis/requirements phase (domain knowledge) and represents it through the framework design language. This is a creative task, and a systematic approach to generate the framework design from requirements level artifacts (use cases and scenarios [42, 43], for example) has not been yet investigated.

The *framework editor* tool is used to assist this task verifying the integrity of the designs. It provides a set of transformations that creates and modifies designs written in the UML-extended design language. Figure 4.2 describes, as an example, the transformations used to create part of a framework extended class diagram. Each line in Figure 4.2 illustrates the use of a different transformation. *Framework editor* provides transformations for defining extended and standard class diagrams, template diagrams, interaction diagrams, and instantiation diagrams.

The system supports the existence of several projects at the same time, each of which represents a different system in the design language.

```
aladin :-
          createProject(aladin),
          createClass(aladin, showCourse, _),
          createMethod(aladin, showCourse, showCourse, [...]),
          createVariationMethod(aladin, showCourse, selectCourse, [...]),
          createMethod(aladin, showCourse, showContent, [...]),
```

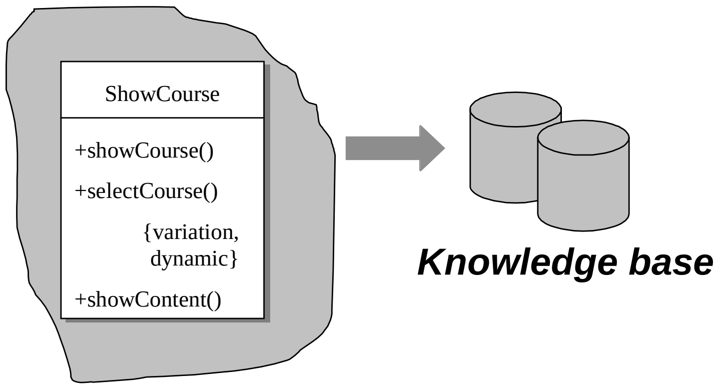
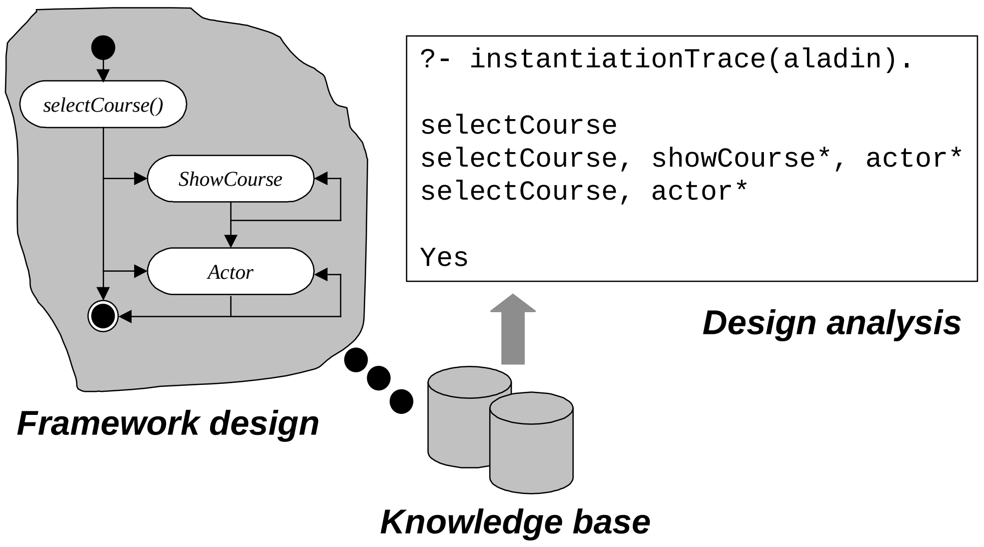


Figure 4.2. Creating a framework design

Framework editor also allows the definition of several design analyses that may be used to help the job of the framework designer. The supported analyses are described in the design analyses meta-artifact, which is interpreted by the tool. This approach allows the incorporation of new analyses into the system at any time.



```
?- instantiationTrace(aladin).

selectCourse
selectCourse, showCourse*, actor*
selectCourse, actor*

Yes
```

Figure 4.3. Design analysis example

Figure 4.3 illustrates an analysis that uses the information in the instantiation diagram to list all the possible instantiation traces of a given framework. Through this design analysis an error in the framework instantiation diagram of the ALADIN framework was detected: the trace *selectCourse(), ShowCourse** is not allowed. If S*howCourse* had been instantiated *Actor* would also have to be. However, these two hot-spots should not be related.

Figure 4.4 shows a high-level architecture of the tool. The design analyses artifact is represented in gray because it is a meta-artifact. There is also the role of system administrator, elided in the figure, which is responsible only for configuring the meta-artifacts.

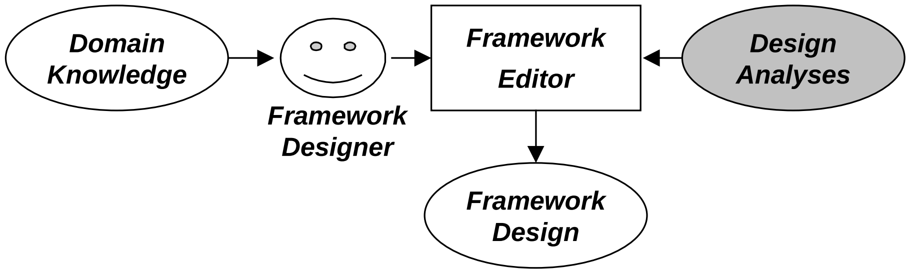


Figure 4.4. Framework editor architecture.

4.3 THE IMPLEMENTATION PHASE

The implementation phase is responsible for generating the actual framework and the framework documentation. There are two tools used in this phase: *code generation* and *documentation*.

Code generation is responsible for mapping the new design elements of the framework design language into appropriate implementation level structures. More specifically, it is responsible for eliminating the variation methods and extension classes from the design, since they do not have a direct implementation level representation. The standard UML artifacts and extension interfaces do not need to be mapped since they have a direct correspondence to implementation level constructs. This mapping is based on meta-artifacts that describe the transformations. These meta-artifacts are the implementation models. Different implementation models define different mappings. The tool supports the definition of new implementation models, allowing experimentation with several approaches for modeling hot-spots.

The transformation illustrated in Figure 3.11 is an example of a mapping supported by the *code generation* tool. The implementation model that supports this transformation describes how dynamic variation methods are modeled by the strategy design pattern [34]. Another approach for implementing this kind of hot-spot is the decorator design pattern [34], as shown in Figure 4.5. Figure 4.6 illustrates the supporting implementation model for the strategy approach while Figure 4.7 illustrates the transformations for the decorator approach.

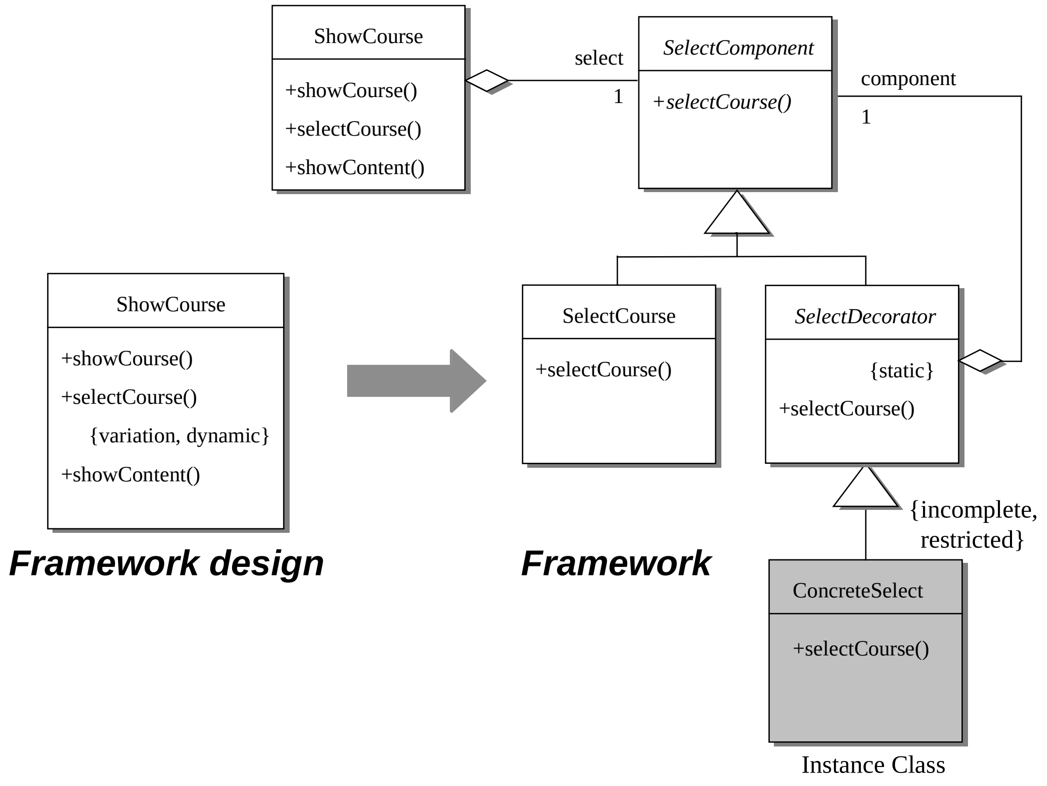


Figure 4.5. Using the decorator design pattern to implement variation methods

```
designPatterns1(Project, NewProject) :-
          [...]
          forall(variationMethod(Project, Class, Method, _),
          strategy(Project, NewProject, Class, Method)),
          [...]

strategy(Project, NewProject, Class, Method) :-
          concat(Method, 'Strategy', NewClass),
          createExtensionInterface(NewProject, NewClass, dynamic),
          createMethod(NewProject, NewClass, Method, public, none, abstract),
          createAggregation(NewProject, Class, NewClass, strategy),
          [...]
```

Figure 4.6. Strategy implementation model

The implementation transformations preserve the design structure described in *Project* and creates *NewProject* to store the generated framework. All the design elements that are not transformed, the kernel elements and the extension interface hot-spots, are copied from *Project* to *NewProject*. The variation method and extension class hot-spots are transformed in the way described by the selected implementation model.

Each valid implementation model artifact has to define at least four transformations: (static and dynamic) variation methods and (static and dynamic) extension classes. Other possible implementation models that have been used to assist framework implementation include different combinations of design patterns, meta-programming [51], aspect-oriented programming (AOP) [50], and subject-oriented programming (SOP) [38], as will be discussed in chapter 6.

```
designPatterns2(Project, NewProject) :-
          [...]
          forall(variationMethod(Project, Class, Method, _),
          decorator(Project, NewProject, Class, Method)),
          [...]

decorator(Project, NewProject, Class, Method) :-
          [...]
          concat(Method, ’Component’, ComponentClass),
          createClass(NewProject, ComponentClass, abstract),
          createMethod(NewProject, ComponentClass, Method, [...]),
          concat(Method, ’Decorator’, DecoratorClass),
          createExtensionInterface(NewProject, DecoratorClass, dynamic),
          createAggregation(NewProject, DecoratorClass, [...]),
          createMethod(NewProject, DecoratorClass, Method, [...]),
          createInheritance(NewProject, ComponentClass, DecoratorClass),
          createInheritance(NewProject, ComponentClass, Class),
          [...]
```

Figure 4.7. Decorator implementation model

All the transformations supported by *code generation* are behavior-preserving, since they map design level elements into implementation level elements while maintaining the framework semantics. It means that the set of applications that can be instantiated within the framework has to be the same before and after the implementation transformation. Figure 4.8 illustrates the tool architecture.

The *documentation tool* is responsible for generating the framework documentation, which is a description of how the framework should be instantiated. This documentation is used to help the application developer during the instantiation phase. The framework documentation can be generated only after the framework is generated, since different implementation models may require different instantiation tasks.

Figure 4.9 shows how the framework documentation varies, depending on the implementation approach chosen. It shows two pattern-based descriptions [45] that describe how to instantiate the *selectCouse()* hot-spot: one for the case that the strategy implementation approach is selected and another for the case in which the decorator approach is selected.

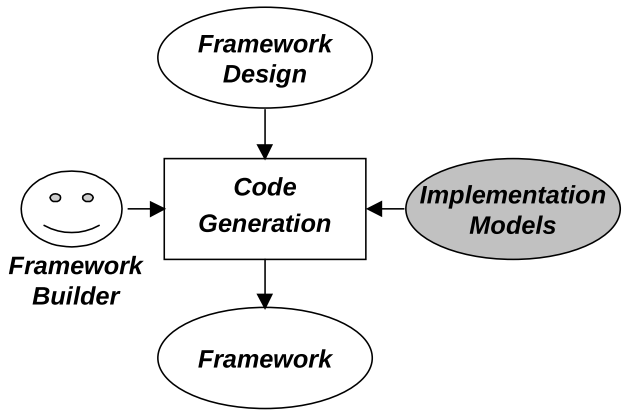
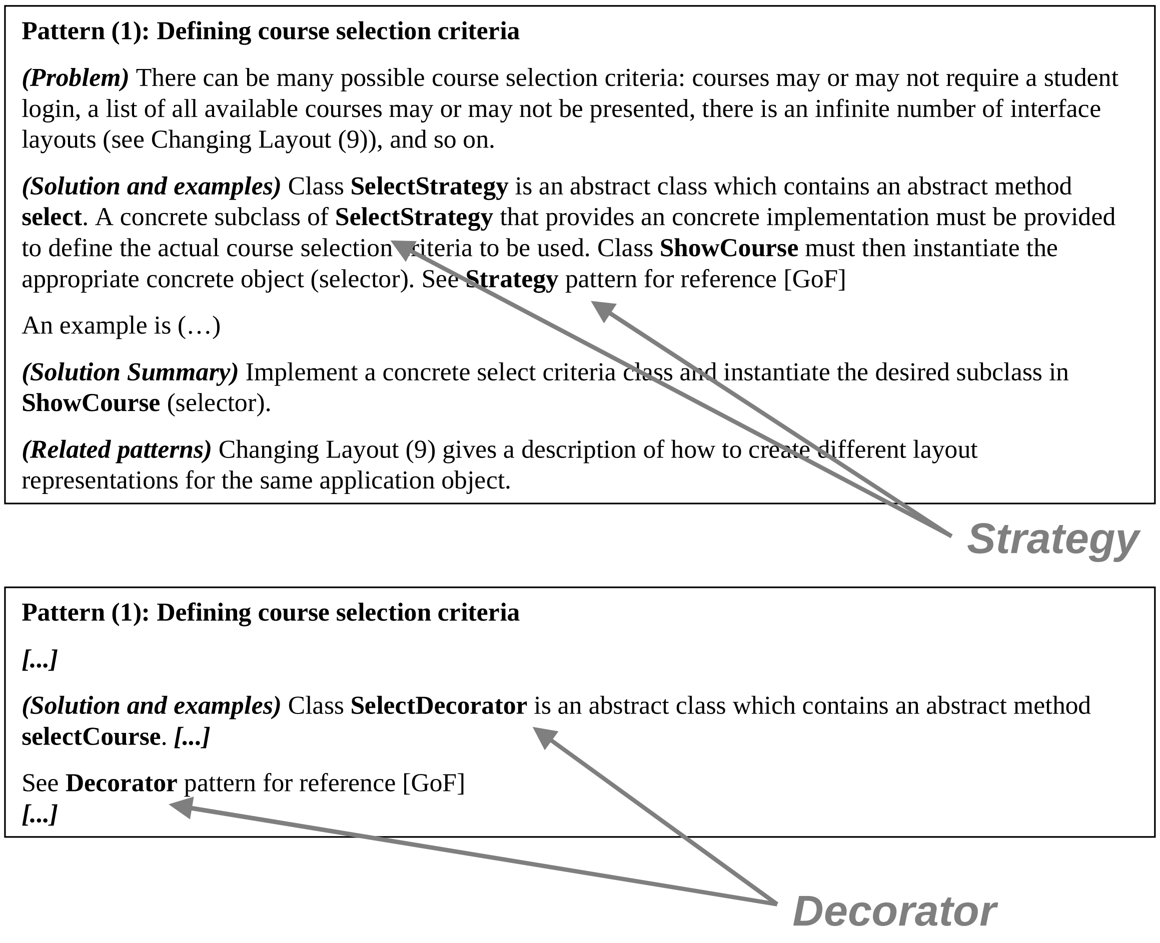


Figure 4.8. Code generation architecture

The tool cannot generate the complete documentation, since several of its fields are informal text that need to be provided by humans. However, as it knows how each hot-spot was implemented, it can automate the generation of some fields. For example, it can provide a default description of how the strategy pattern should be instantiated for the hot-spots that use this approach. An interesting aspect of the *documentation tool* is that it assures that documentation is generated for all of the framework hot-spots that are modeled by the design language.

The *documentation tool* allows for several documentation approaches to be used. The meta-artifact documentation models define the transformations that generate each of the possible documentation approaches from the framework structure. Chapter 6 discusses several techniques that can be used to document frameworks and describes their documentation models. The tool architecture is illustrated in Figure 4.10.



Pattern (1): Defining course selection criteria

(Problem) There can be many possible course selection criteria: courses may or may not require a student login, a list of all available courses may or may not be presented, there is an infinite number of interface layouts (see Changing Layout (9)), and so on.

(Solution and examples) Class **SelectStrategy** is an abstract class which contains an abstract method **select**. A concrete subclass of **SelectStrategy** that provides an concrete implementation must be provided to define the actual course selection criteria to be used. Class **ShowCourse** must then instantiate the appropriate concrete object (selector). See **Strategy** pattern for reference [GoF]

An example is (…)

(Solution Summary) Implement a concrete select criteria class and instantiate the desired subclass in **ShowCourse** (selector).

(Related patterns) Changing Layout (9) gives a description of how to create different layout representations for the same application object.

Pattern (1): Defining course selection criteria

[...]

(Solution and examples) Class **SelectDecorator** is an abstract class which contains an abstract method **selectCourse**. ***[...]***

See **Decorator** pattern for reference [GoF]
[...]

Figure 4.9. Two pattern-based descriptions for two different implementations of the same hot-spot

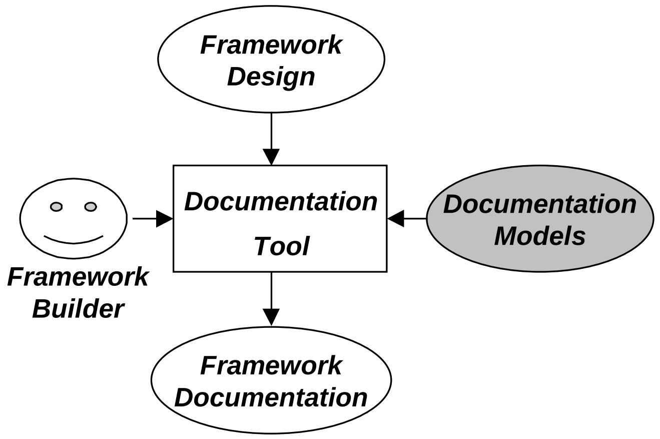


Figure 4.10. Documentation tool architecture

4.4 THE INSTANTIATION PHASE

During the instantiation phase the application developer creates applications from the framework. There are two tools that assist this task: the *process-based instantiation tool* and the *DSL generator*. Both of them depend on the framework definition but they work quite differently.

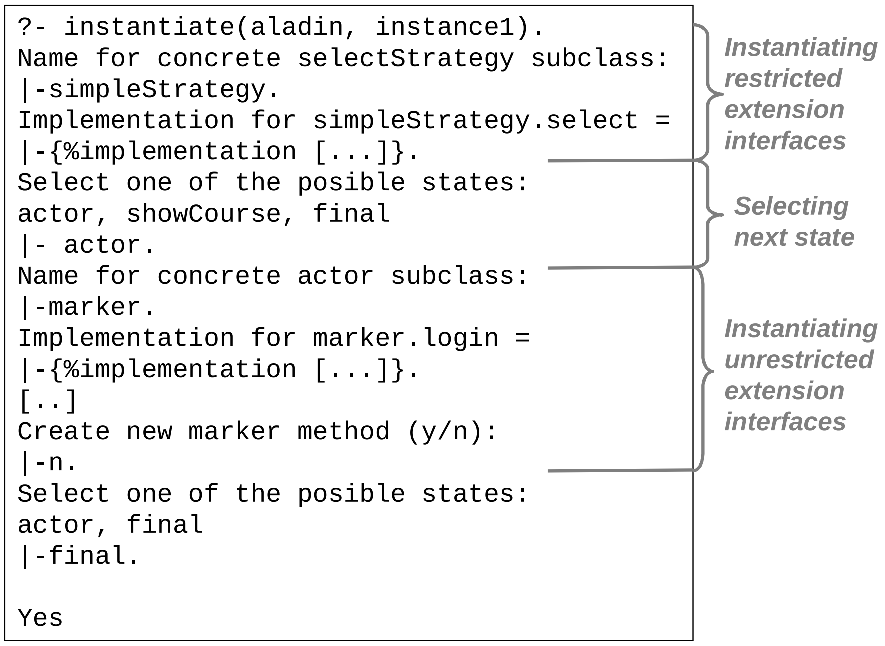
The *process-based instantiation tool* "executes" the instantiation diagrams, which can be seen as a formal description of the framework instantiation process. Each node of the diagram represents one framework hot-spot that has to be instantiated.

The tool begins at the initial state, and moves throughout the diagram until the final state is reached. For each state it visits, it instantiates the hot-spot and moves to the next available state. When more than one next state is available, which is generally the case, it asks the application developer to which state he or she wants to move to.

The frameworks that might be instantiated are only allowed to have extension interface hot-spots. The tool knows what are the exact procedures to instantiate extension interfaces: it has to create a new subclass, ask for the implementation of each of the interface methods, and ask for the definition (signature and implementation) of new methods that might be added. In the case of restricted extension interfaces this last step is not performed. The tool prompts the application developer about all the required information to complete the missing hot-spot information. Figure 4.11 illustrates a sample instantiation for the ALADIN framework [31]. Every time a framework is instantiated using the tool a new project is created *(instance1*, in this example).

Note that depending on the implementation model selected, different instantiation tasks may be required for the same hot-spot. The *process-based instantiation tool* can be seen as a wizard [10, 63] that assists the framework instantiation. Figure 4.12 illustrates the tool architecture.

The *DSL generator* generates a description of a DSL from the framework design representation. The idea is to define a language that syntactically assures that the framework instantiation restrictions are preserved. Based on this approach, the framework instantiation consists in programming in the framework DSL. This program is then compiled to complete the missing hot-spot information, generating the desired application.



```
?- instantiate(aladin, instance1).
Name for concrete selectStrategy subclass:
|-simpleStrategy.
Implementation for simpleStrategy.select =
|-{%implementation [...]}.
Select one of the posible states:
actor, showCourse, final
|- actor.
Name for concrete actor subclass:
|-marker.
Implementation for marker.login =
|-{%implementation [...]}.
[..]
Create new marker method (y/n):
|-n.
Select one of the posible states:
actor, final
|-final.

Yes
```

Instantiating restricted extension interfaces
Selecting next state
Instantiating unrestricted extension interfaces

Figure 4.11. Process-based instantiation example

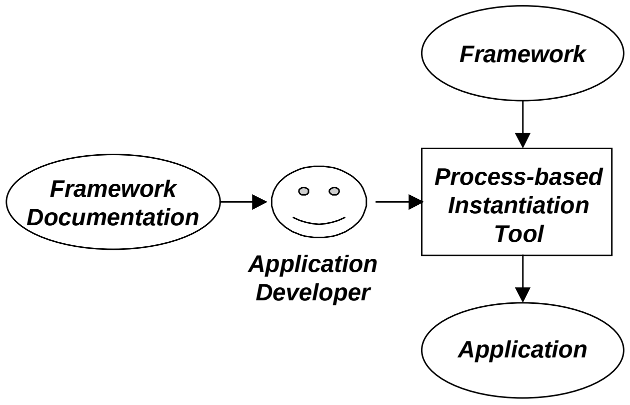
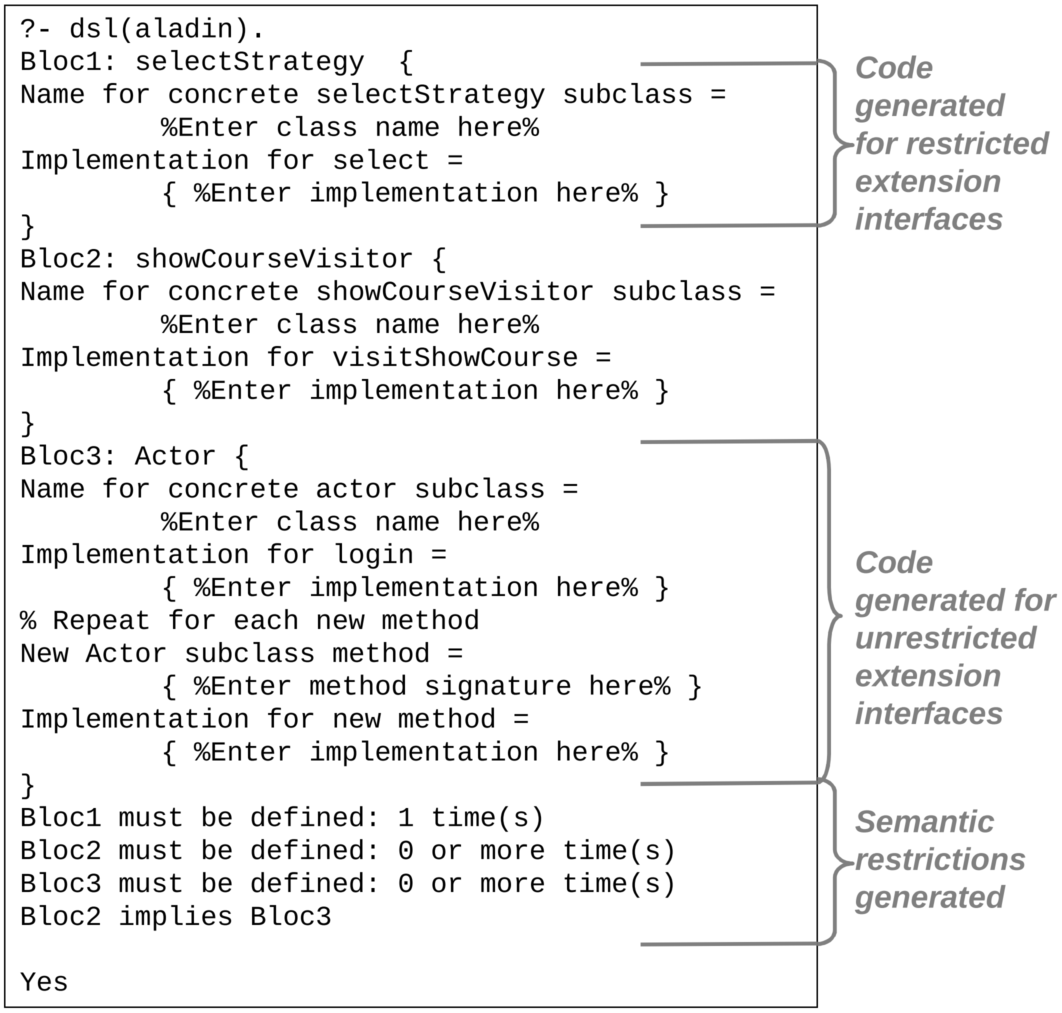


Figure 4.12. Process-based instantiation tool architecture

Figure 4.13 shows a description of a language that may be used to instantiate the framework modeled in Figure 3.8. *DSL generator* uses the information on how to instantiate extended interfaces and the representation of the instantiation diagrams to generate the language description.

The DSL compiler is responsible for realizing semantic checks and generating appropriate error reports. In this example, if a given program does not specify *Bloc1* an error should be generated: “You have to define one course selection strategy”, for example. The compiler can also verify if the semantic restrictions defined by the OCL specifications and template diagrams are being preserved. However, doing so may be very complex since checking program properties is computationally undecidable in general.



```
?- dsl(aladin).
Bloc1: selectStrategy  {
Name for concrete selectStrategy subclass =
        %Enter class name here%
Implementation for select =
        { %Enter implementation here% }
}
Bloc2: showCourseVisitor {
Name for concrete showCourseVisitor subclass =
        %Enter class name here%
Implementation for visitShowCourse =
        { %Enter implementation here% }
}
Bloc3: Actor {
Name for concrete actor subclass =
        %Enter class name here%
Implementation for login =
        { %Enter implementation here% }
% Repeat for each new method
New Actor subclass method =
        { %Enter method signature here% }
Implementation for new method =
        { %Enter implementation here% }
}
Bloc1 must be defined: 1 time(s)
Bloc2 must be defined: 0 or more time(s)
Bloc3 must be defined: 0 or more time(s)
Bloc2 implies Bloc3

Yes
```

Code generated for restricted extension interfaces
Code generated for unrestricted extension interfaces
Semantic restrictions generated

Figure 4.13. DSL description example

Note that some parts of the DSL ask for method implementation. These parts may be completed with methods written in the language in which the framework is implemented or in any other language. In the first case the method does not need to be transformed and may be directed attached to the framework code. If another language is used, on the other hand, the method has to be transformed before being used. The advantage of using a language other than the framework language is that simpler languages may be used for hot-spots that perform simple tasks. If a variation method only displays a web-page, for example, HTML may be used to specify its behavior. If it only queries a database then SQL can be used. It may be the case that a different language is used for each different hot-spot in the same framework. In this case the DSL can be seen as a “weaver” in the aspect-oriented programming (AOP) [50] point of view (AOP is briefly described in section 6.1.3). The selection of the most appropriate language for implementing each hot-spot is a creative task and cannot be supported by a tool.

Figure 4.14 shows the architecture of the *DSL generator tool*. The actual DSL compiler is not generated in the current implementation of the tool, only the language syntax and the description of semantic restrictions that should be verified are generated. Note that in the case of run-time reconfiguration the DSL compiler works exactly as an MOP [51] .

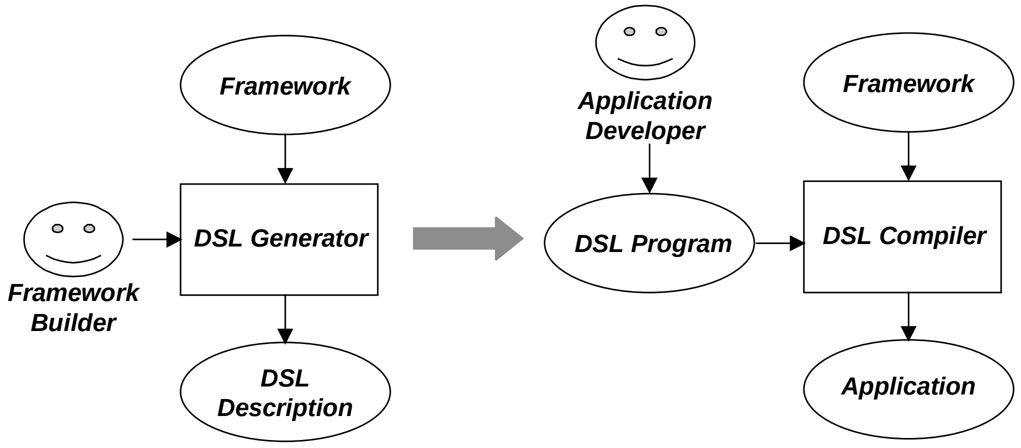


Figure 4.14. DSL generator architecture

4.5 THE MAINTENANCE PHASE

There are two complementary approaches that are used by the *maintenance tool:* refactoring [47, 66] and unification [2] rules.

Refactorings are horizontal behavior-preserving transformations that may be applied at the design and implementation levels. Examples of refactorings are moving common behavior to super classes, renaming methods, and so on.

An example of a design-level refactoring applied to the design represented in Figure 3.8 is the division of the *showContent()* method into two complementary methods: *getContent()* and *displayContent(),* as shown in Figure 4.15. This is a valid design refactoring if the semantics of the method is preserved. Since this cannot be automatically verified, the framework maintainer has to interact with the tool to guarantee that the transformation may be applied. This refactoring can be useful, for example, if the *getContent()* method is used in other places in the same system.

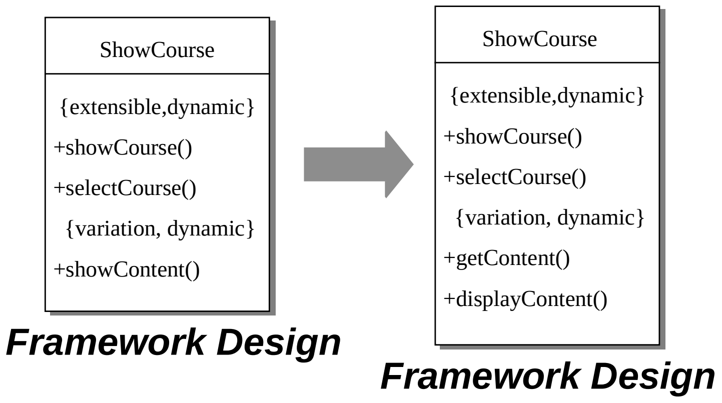


Figure 4.15. Splitting method refactoring

Besides applying refactorings, the *maintenance tool* can also determine when they should be applied. Examples of analyses of whether refactorings should be applied include:

- List the methods that can be moved to abstract super classes. If a given method exists in two different classes, a common abstract super class may be created to store it;
- Check if the interface of a given class can be minimized. If the class has public methods that are unused by other classes in the system they can be transformed into private methods.

In both examples the checks can be completely performed by the tool, without the need for human support. However, it may be the case that human support is necessary in order to complete some verifications that might be needed.

Unification rules define a way to incorporate new abstractions into the framework structure. These new abstractions are represented by the domain changes artifact in Figure 4.1. As in refactoring, unification rules are transformations on the framework design structure. Unlike refactoring, unification rules are not behavior-preserving: the semantics of the framework is changed to incorporate the semantics of the new features.

Considering the design presented in Figure 3.8, a unification situation exists if variations of the *showContent()* implementation are required by a given framework instance. Since *showContent()* is not part of the framework hot-spot structure, the instance would have to violate the framework architecture to redefine *showContent()*.

Figure 4.16 illustrates a unification procedure used to avoid this problem, in which the domain changes artifact is represented by a design where the *showContent()* method semantics differs from the one previously supported by the framework. Note that some hot-spot information cannot be automatically generated and has to be provided by the framework maintainer.

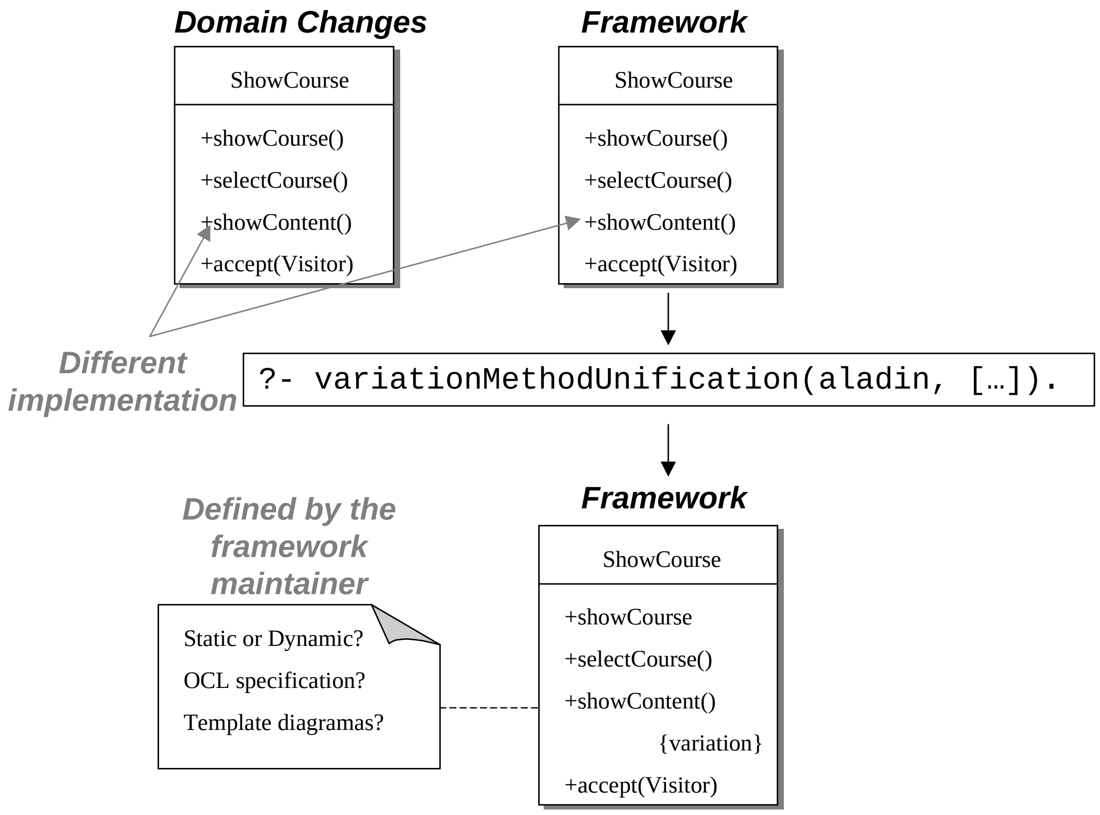


Figure 4.16. Unification rule example

Sometimes the application of refactoring procedures before the unification can improve its result. In this same example, it might be the case that the *showContent()* method varies only when displaying the course content but is stable when getting the content. Then, if the splitting refactoring had been applied before the unification the variation hot-spot would be *displayContent()* only, and not the entire *showContent()* method.

4.6 SUMMARY

A transformational approach for the framework development process is presented to illustrate how tools can automate several tasks if frameworks are represented through the design language proposed in chapter 3.

The design phase is responsible for modeling the frameworks using the language and for performing design analysis on the framework design structure. The implementation phase is responsible for mapping variation methods and extension classes into existing programming language constructs. The instantiation phase assists the framework user in the process of creating valid applications based on the framework. Finally, the maintenance phase supports framework restructuring through refactoring [47, 66] and unification [2] rules.

Chapter 5 – Framework Design

This chapter describes the framework editor tool to illustrate how tools that support the framework design language as the underlying design model representation may be defined. It shows the transformations used to create extended class diagrams, template diagrams, and instantiation diagrams. These transformations validate the design integrity and store its artifacts into the system knowledge base. The support for design analyses is also discussed.

5.1 FRAMEWORK EDITING

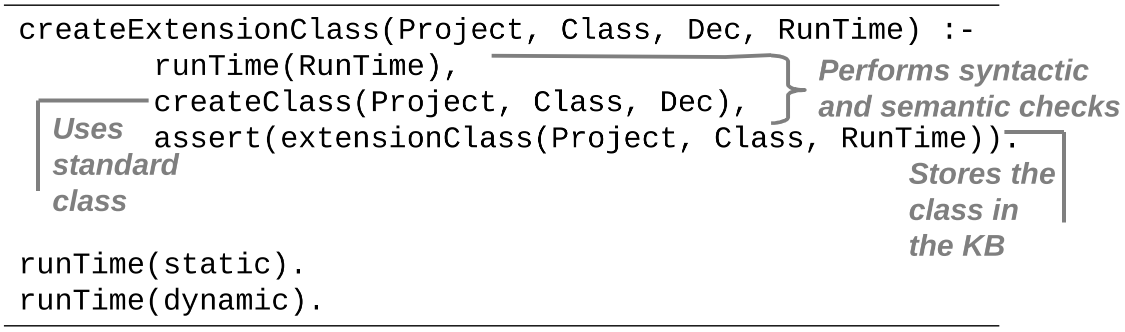
The *framework editor* tool uses the UML-extended design language described in chapter 3 as the underlying representation language for the framework design artifacts. The framework designer uses this tool to define the design diagrams. There are five kinds of diagrams supported by the design language: standard class diagrams, extended class diagrams, interaction diagrams, template diagrams, and instantiation diagrams. The design elements are defined in the system through transformations:

```
createClass(Project, Class, Dec) :-
       project(Project),                      } Performs syntactic
       not(class(Project, Class)),            } and semantics
       classDeclaration(Dec),                 } checks
       assert(class(Project, Class)),                        } Stores the
       assert(classDeclaration(Project, Class, Dec)).        } class in
                                                             } the KB
classDeclaration(none).
classDeclaration(final).
classDeclaration(abstract).
```

Figure 5.1. Creating classes

Standard UML class diagram transformations are used to generate the standard class diagram elements, such as interfaces, classes, relationships, methods, and attributes. Transformations that define OCL specifications and method implementations are also included in this category, since they are represented through notes attached to class diagram elements. The current version of the *framework editor tool* does not check the syntax of the OCL specifications and method implementations: both are treated as plain text strings. A transformation for creating classes is presented in Figure 5.1 as an example.

Extended class diagram transformations are used to create variation methods, extension classes, and extension interfaces. These elements are defined on top of standard UML elements. The variation methods are based on the definition of standard methods while the extension classes and interfaces are based on the definition of standard UML classes. Figure 5.2 illustrates the transformation for creating extension classes. Note that it uses the transformation that creates standard classes;



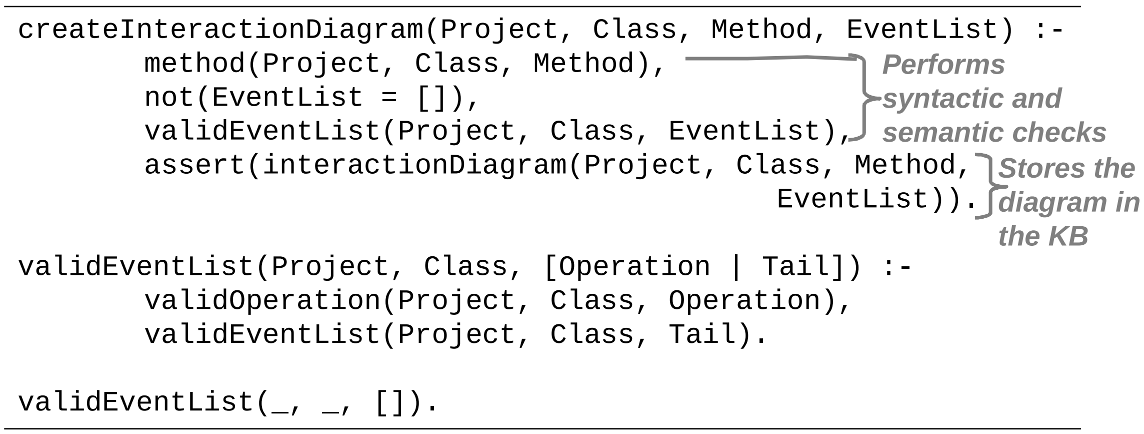
```
createExtensionClass(Project, Class, Dec, RunTime) :-
        runTime(RunTime),
        createClass(Project, Class, Dec),
        assert(extensionClass(Project, Class, RunTime)).

runTime(static).
runTime(dynamic).
```

Performs syntactic and semantic checks
Uses standard class
Stores the class in the KB

Figure 5.2. Creating extension classes

UML interaction diagrams and are created through the transformation described in Figure 5.3.



```
createInteractionDiagram(Project, Class, Method, EventList) :-
        method(Project, Class, Method),
        not(EventList = []),
        validEventList(Project, Class, EventList),
        assert(interactionDiagram(Project, Class, Method,
                                                EventList)).

validEventList(Project, Class, [Operation | Tail]) :-
        validOperation(Project, Class, Operation),
        validEventList(Project, Class, Tail).

validEventList(_, _, []).
```

Performs syntactic and semantic checks
Stores the diagram in the KB

Figure 5.3. Creating interaction diagrams

Template diagrams extend the definition of the standard interaction diagrams with the *{optional}* tagged value to represent events that may occur depending on the hot-spot instantiation. The transformation to create template diagrams is very similar to the one used to create standard interaction diagrams and for this reason will not be illustrated.

Instantiation diagrams represent the activity diagrams that model the framework instantiation process. To define instantiation diagrams using the *process-based instantiation tool* the framework designer has to first create the states of the diagram and then create the transitions. There are four possible kinds of states: variation method, extension class, extension interface, and restricted extension interface. When variation methods and extension classes are transformed during the implementation phase their respective states must also be. The two kinds of extension interface states are used during the instantiation phase. Figure 5.4 shows the transformations to create variation method states and transitions.

```
              createMethodState(Name, Project, Class, Method) :-
      Performs   variationMethod(Project, Class, Method, _),
syntactic and    not(state(Project, Name)),
     semantic    assert(methodState(Name, Project, Class,
       checks                                     Method)),  Stores the
                 assert(state(Project, Name)).                 diagram in
                                                               the KB

              createTransition(Project, Source, Target) :-
                 state(Project, Source),       Performs
                 state(Project, Target),       syntactic and
                 not(Source = final),          semantic checks
                 not(Target = initial),
                 assert(transition(Project, Source,   Stores the
                                         Target)).    diagram in
                                                      the KB
              state(_, initial).
              state(_, final).
```

Figure 5.4. Creating instantiation diagrams

Framework editor allows several designs to be defined at the same time. Different designs are stored as different *projects* in the system knowledge base. The definition of projects by the framework designer is based on text files that are interpreted by the tool. Figure 5.5 illustrates this architecture.

Each project file must define a predicate with the same name as the file name. The tool reads the file and executes this predicate to load the project definition into the system's knowledge base.

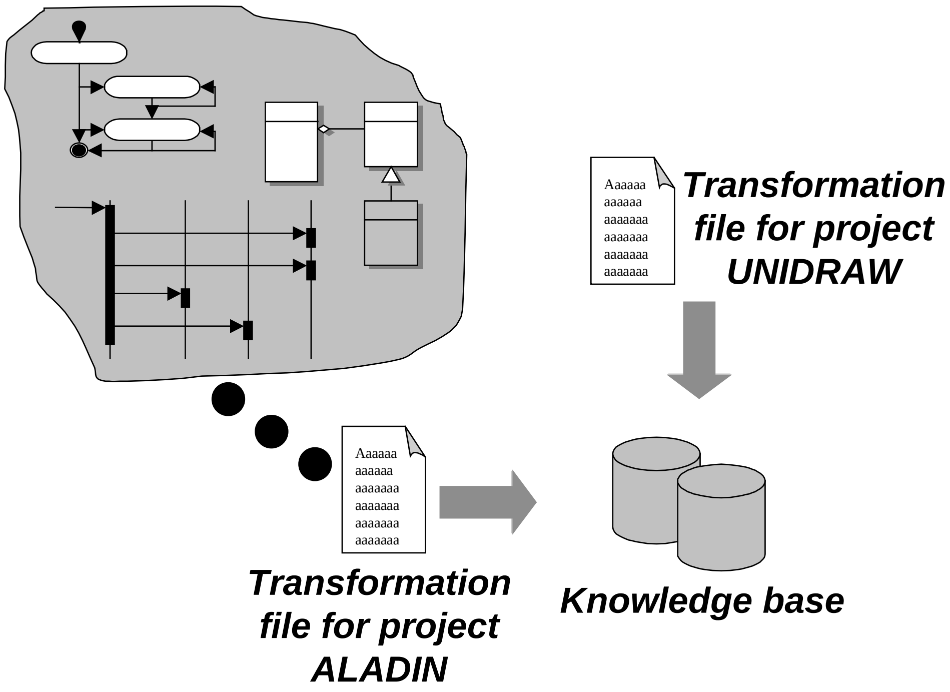


Figure 5.5. Framework editor: support for multiple projects

5.2 DESIGN ANALYSES

Framework editor allows for the definition of several design analyses, which are useful to allow for systematic reasoning about the framework structure. The design analyses are defined in the system through the meta-artifact design analyses, which is a PROLOG file that is interpreted by the tool.

```
listMethods(Project, Class) :-
       forall(method(Project, Class, Method),
       listMethod(Project, Class, Method)).

listMethod(Project, Class, Method) :-
       methodAccess(Project, Class, Method, Access),
       methodStatic(Project, Class, Method, Static),
       methodDeclaration(Project, Class, Method, Dec),
       Term =.. [Method, Access, Static, Dec],
       write_ln(Term),
       listSignature(Project, Class, Method),
       listImplementation(Project, Class, Method).
```

Cannot change the knowledge base

Figure 5.6. Design analysis example

Any query about the system knowledge base that does not change the content of the stored projects is a valid design analysis. Examples of analyses vary from simple project listing to complex statistics regarding hot-spot interdependencies.
Currently, there are several design analyses defined, including project listing, verification of instantiation traces (as shown in Figure 4.3), checking if interaction diagrams can be template diagram instances, and so on. Figure 5.6 illustrates an analysis that lists the methods of a class. Note that it does not modify the software artifacts stored in the knowledge base.

5.3 RELATED WORK

There are several UML case tools such as Rational Rose (http://www.rational.com) and Visio Enterprise (http://www.visio.com). As in *framework editor* these tools provide semantic checks that may be applied on the UML diagrams. Unlike in *framework editor* these tools provide nice visual interfaces for editing the design models by direct manipulation. However, they fail to support design concepts such as the ones proposed in this work. This makes it impossible for them to perform design analysis on the framework structure since there is no way they can distinguish the kernel subsystem from the hot-spots.
Several design pattern tools [12, 24, 29, 61] have been proposed to facilitate the definition of design patterns, to allow the incorporation of patterns into specific projects, to instantiate design descriptions, and to generate code. However, they leave the selection of the most appropriate pattern to model each framework hot-spot in the hands of the framework designer. Although this is obviously a creative task, one of the most important claims of this thesis is that frameworks should first be modeled through an appropriate design language (as the one proposed in chapter 3) and the selection of design patterns, or other possible techniques, to model the hot-spots should be left to later steps in the development process. Based on this approach, tools that assist the systematization of the selection of the most appropriate modeling technique may be constructed, simplifying the job of the framework designer. This point will be discussed in more detail in chapter 6, when several design-implementation mappings are discussed.

5.4 SUMMARY

The transformations used to create a framework design may be classified into five categories: standard class diagrams, extended class diagrams, standard interaction diagrams, template diagrams, and interaction diagrams. Several design analyses may be added to *framework editor*, varying from simple listing procedures to complex interdependency analyses. These analyses must not alter the content of the knowledge base, which stores all the project definitions.

Several tools that support OO design have been proposed, such as Rational Rose (http://www.rational.com) and Visio Enterprise (http://www.visio.com). However, none of them can provide adequate support for the framework designer since their underlying representation languages do not explicitly model the framework hot-spots. Even when design patterns are considered as first-class citizens [12, 24, 29, 61] the framework designer still has no support for selecting the most appropriate solution to model each hot-spot.

If the design language proposed in this work is used, the framework designer can work at a higher level of abstraction, focusing on what are the domain hot-spots and not on how they should be implemented. The selection of the best modeling technique for each hot-spot can be assisted by a tool, at the implementation stage. Finally, design analysis on hot-spot structures also cannot be performed if the underlying representation language does not explicitly represent them.

Chapter 6 – Framework Implementation

This chapter presents several techniques that can be used to implement the framework hot-spots and generate the corresponding documentation. These techniques are defined as transformations on the framework design representation. Transformations that map each different kind of hot-spot into appropriate implementation level constructs are defined by the implementation models meta-artifacts, which are interpreted by the code generation tool. Transformations that guide the generation of framework documentation from the hot-spot definitions are defined by the documentation models meta-artifacts, which are interpreted by the documentation tool.

6.1 IMPLEMENTATION MODELS

This section shows how the proposed design language can assist the implementation phase of a given framework development process by highlighting important aspects of the hot-spots that must be considered for the selection of appropriate techniques to model them.

The main purpose of this section is to show that the design language can make the implementation process more systematic. It does not attempt to prescribe the best implementation technique to be used with frameworks. The implementation models are used by the *code generation* tool to map the variation methods and extension classes into implementation level constructs.

6.1.1 DESIGN PATTERNS

Most published design patterns [13, 34, 95] are implementation techniques for variation and extension problems. This subsection shows how the framework design language proposed in chapter 3 can help the selection of the best design patterns to implement each framework hot-spot. Although this is a creative task that cannot be completely automated, all the information needed to accomplish it is documented in the design artifacts produced at the design stage (extended class diagrams and template diagrams). The analysis of these artifacts can assist in narrowing the search for an appropriate pattern. The use of abstract coupling patterns [71] for implementing variations methods and the visitor design pattern [34] for implementing extension classes are examples of appropriate pattern implementations for hot-spots. Examples of design patterns based on abstract coupling include abstract factory, builder, command, interpreter, observer, state, and strategy [34, 71].

Figure 6.1 illustrates an example of this approach. The strategy design pattern is used to implement the dynamic variation *selectCourse()***,** and the visitor design pattern is used to implement the class extension *ShowCourse*. Note that Figure 6.1 is an *implementation diagram* that uses design patterns to model the hot-spots of the *design diagram* shown in Figure 3.8. The use of the decorator design pattern [34] to implement variation method hot-spots is another possible approach, as shown in Figure 4.5.

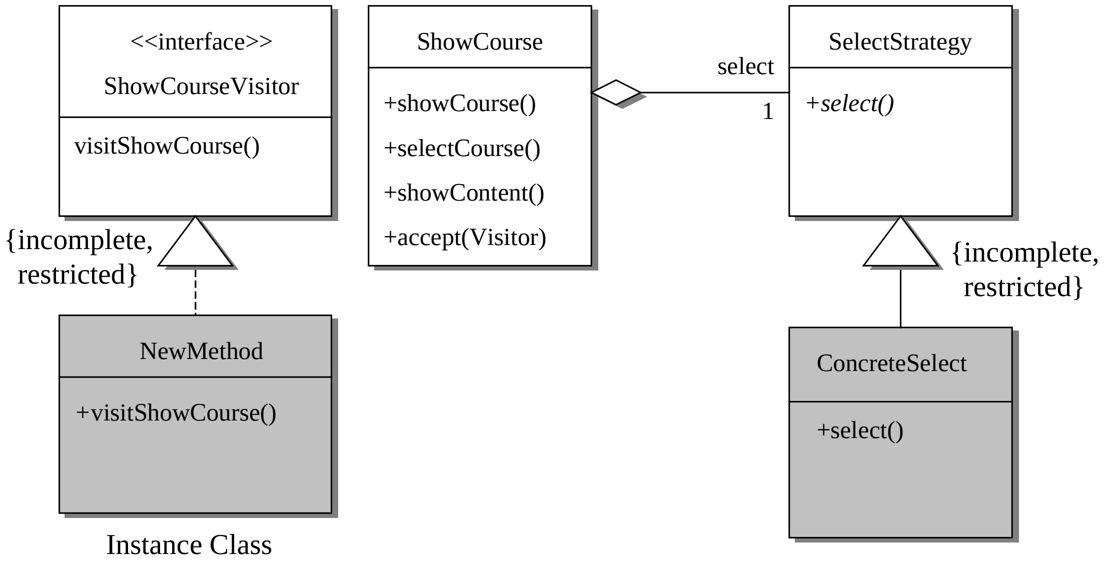


Figure 6.1. Using design patterns to model framework hot-spots

Implementing hot-spots with the selected design patterns does not guarantee that the OCL specifications and the template diagrams (Figure 3.9(a)) will be satisfied. Strategy, decorator, and visitor are white-box patterns, and do not impose any restrictions on their implementation. The use of the template design pattern [34], for example, may help in satisfying the constraints on the design specified by template diagrams. Although the selection of the most appropriate design pattern is a creative task, automatic selection of patterns to implement each kind of hot-spot can be achieved by generating a standardized implementation of the framework from its design representation.

Note that a framework user who has access to the framework design representation and the description of design patterns that were used to implement the hot-spots can easily discover how the instantiation process should proceed (which classes to inherit from, which methods to override, and so on).

The implementation models are also responsible for mapping the instantiation diagrams, converting the variation method and extension class states into extension interface-states. Figure 6.2 shows this transformation while Figure 6.3 illustrates the implementation model that describes the variation method-strategy and extension class-visitor transformations.

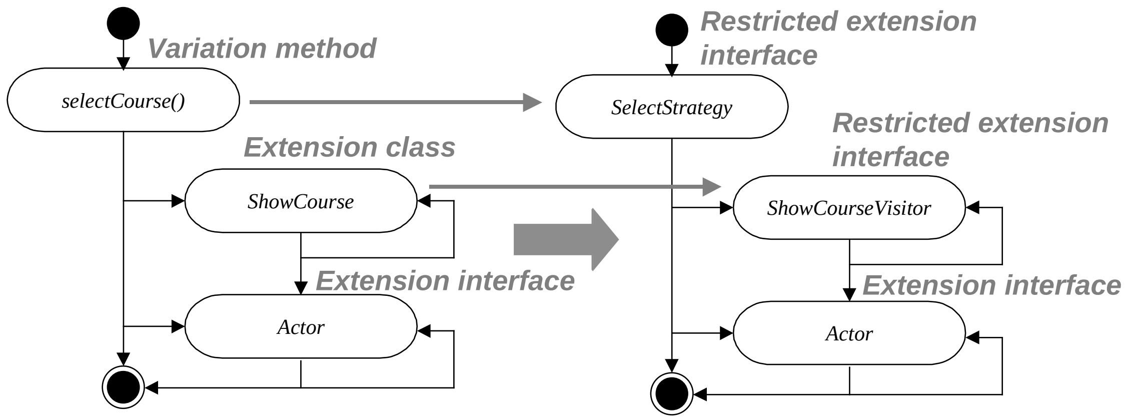


Figure 6.2. Instantiation diagram conversion

```
designPatterns(Project, NewProject) :-
     (a)copyUMLElements(...),
     (b)forall(variationMethod(...), strategy(...)),
     (c)forall(extensionClass(...), visitor(...)),
     (d)forall(extensionInterface(...), copyExtensionInterface(...)),
     (e)forall(transition(...), createTransition(...)),
        [...]
```

(a) Copy the standard kernel elements to the new project
(b) Model variation methods using strategy and converts variation method states
(c) Model extension classes using visitor and converts extension class states
(d) Copy the extension interfaces
(e) Copy instantiation diagram transitions

Figure 6.3. Design pattern-based implementation model

6.1.2 META-LEVEL PROGRAMMING

Meta-level programming [51], as discussed in section 3.1.2, is a suitable technique for implementing dynamic variation method hot-spots. The design notation proposed in this work facilitates the construction of meta-object protocols since all the required information is provided by the OCL specifications and by the template diagrams.

The development of the MOP proposed in Figure 3.3 for the *selectCourse()* hot-spot can profit from the template diagram shown in Figure 3.9(a). In cases in which more than one template diagram is provided the MOP has to be consistent with all of them.

The documentation at the design level of hot-spot instantiation restrictions is all the information required for the development of the MOP. If the constraints on the variations are not specified the MOP development becomes much harder since there is no information about which instances should be provided by the MOP. The transformations in Figure 6.4 show how the MOP specification can be partially derived from the design notation.

```
mop(Project, NewProject) :-
          [...]
          forall(variationMethod(Project, Class, Method, _),
          mop(Project, NewProject, Class, Method)),
          [...]

mop(Project, NewProject, Class, Method) :-
          concat(Class, ’MOP’, NewClass),
          createClass(NewProject, NewClass, none),
          concat(’edit’, Method, NewMethod),
          createMethod(NewProject, NewClass, NewMethod, [...]),
          createAggregation(NewProject, Class, NewClass, [...]),
          askImplementationMop(NewProject, NewClass, NewMethod, Imp),
          [...]
```

Figure 6.4. MOP implementation model

The predicates in Figure 6.4 are similar to the ones used for the strategy pattern, but the user has to provide the implementation for the MOP *(askImplementationMOP).* The tool guides the user in the process of providing the MOP specification by showing the template diagrams and OCL specifications that have to be preserved by the protocol.

The MOP implementation model does not use extension interfaces to model the variation methods. Consequently, the variation methods do not need to be instantiated before run-time. That is why these hot-spots are eliminated from the instantiation diagram, as shown in Figure 6.5.

The approaches to implementing hot-spots that are based on extension interfaces require class inheritance and method overriding and have to be instantiated at compile-time. The design pattern transformations are included in this category [34].

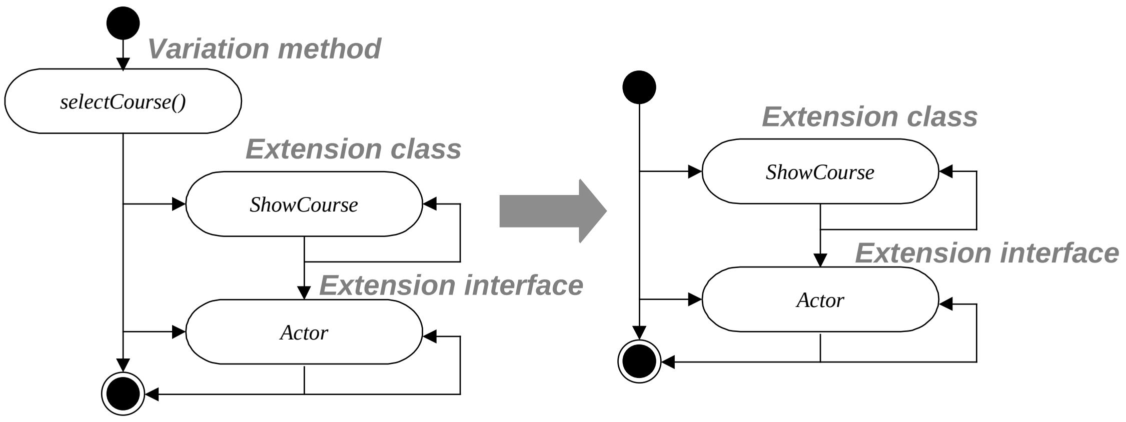


Figure 6.5. Eliminating variation method states

6.1.3 ASPECT-ORIENTED PROGRAMMING

"Aspects" are cross-cutting[6] non functional constraints on a system, such as error handling and performance optimization. Current programming languages fail to provide good support for specifying "aspects," and consequently the code that implements them is typically very tangled and spread throughout the entire system. Aspect-oriented programming (AOP) [50] is a technique proposed to address this problem. An application that is based on the AOP paradigm has the following parts: (i.a) a component language, used to program the system components, (i.b) one or more aspect languages, used to program the aspects, (ii) an aspect "weaver", which is responsible for combing the component and the aspect languages, (iii.a) a component program that implements the system functionality using the component language, and (iii.b) one or more aspect programs that implement the aspects using the aspect languages.

As discussed throughout the dissertation, the main idea behind frameworks is not the specification of aspects, but the modeling of the variability and extensibility requirements of a domain. However, AspectJ [57], which is an AOP extension for Java, can be seen as

[6] If there are two concepts that are better represented in different programming languages (such as code optimization and the logic of the system itself) they are said to be cross-cutting concepts [50].

a general purpose development tool that allows the definition of the program and its aspects in Java.

AspectJ allows the addition of code before or after a method or a class constructor is executed (keywords *before* and *after)*. The language also allows the addition of code using *catch* and *finally* (similar to Java's catch and finally constructs). All of these keywords determine the points in the component program at which the aspect codes, also written in Java, should execute. AspectJ also provides the *new* construct, for extending classes with new elements specified in separate aspects.

Even though the primary concern of AspectJ is the specification of non-functional aspects, such as code optimization, it can be used in a straightforward way for implementing variation and extension hot-spots.

Figure 6.6 is an example of how AspectJ can be used to implement framework variations and extensions. Aspect *TipOfTheDay* implements a method *showTip(),* which is introduced (keyword *new)* to the *ShowCourse* class whenever this aspect is plugged into the system. Aspects are plugged in by invoking the weaver as shown next.

```
% ajweaver ShowCourse.ajava TipOfTheDay.ajava SelectCourseOption2.ajava
```

The approach for variations is similar: all the different implementations of a given variation are placed in different aspects *(SelectCourseOption1* and *SelectCourseOption2,* for example). When instantiating the framework, one aspect that implements each variation must be plugged-in. In this example, the variation method will execute right after the *ShowCourse* class constructor executes.

```
aspect TipOfTheDay {
            static new void ShowCourse.showTip() { // implementation }
            }
aspect SelectCourseOption1 {
            static after void ShowCourse(*) { // implementation of option 1}
}
aspect SelectCourseOption2 {
            static after void ShowCourse(*) { // implementation of option 2}
}
```

Figure 6.6. Using AspectJ to implement framework variations and extensions

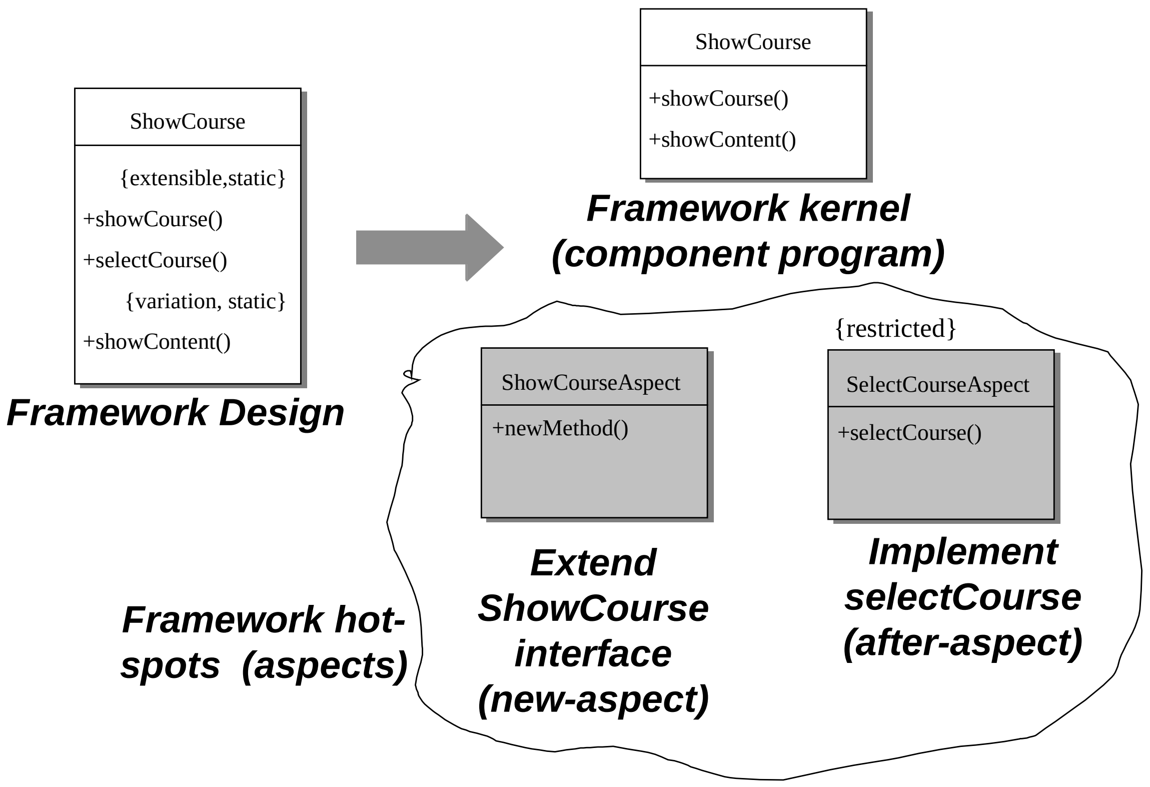


Figure 6.7. AOP-based implementation transformation

Figure 6.7 illustrates the implementation transformations based on the AOP approach for implementing hot-spots. The framework kernel is implemented as the component program while each hot-spot is implemented in a separate aspect. The aspects are represented as instantiation classes since they exist only in the instantiated applications. The restricted constraint is applied to the *SelectCourseAspect* to indicate that in those aspects only the *selectCourse()* method should be defined. The instantiation of aspects can be done using exactly the same procedure used to instantiate extension interfaces. The instantiation diagram transformation is shown in Figure 6.8. The AOP implementation model is described in Figure 6.9

Since extensions and adaptations require invocation of the weaver, this approach is valid only for static hot-spots (note that the definition of the run-time requirements of the two hot-spots in the framework design presented in Figure 6.7 have been changed to static). In addition, the current implementation of AspectJ is a beta version, and the approach has not yet been used in large scale applications. However, all the information required for implementing and instantiating the aspects can be derived from the framework design diagrams.

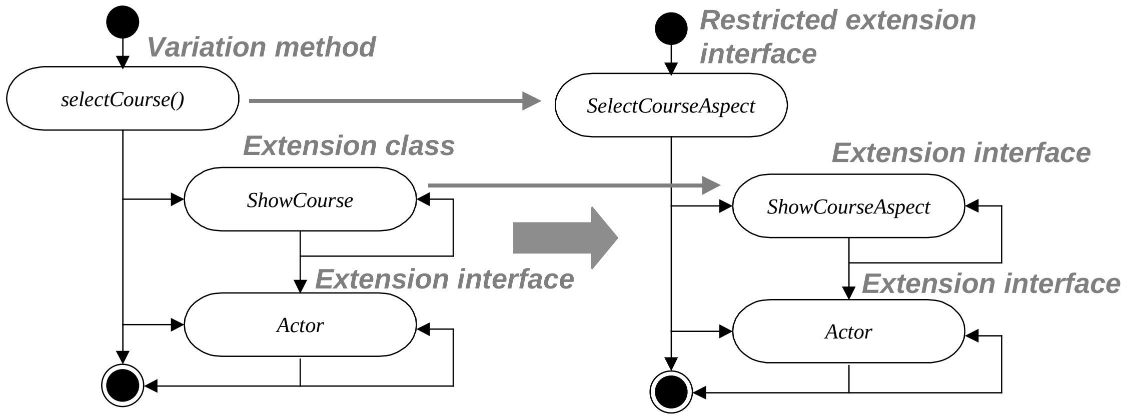


Figure 6.8. Aspects as extension interfaces

```
aop(Project, NewProject) :-
     (a)copyUMLElements(...),
     (b)forall(variationMethod(...), afterAspect(...)),
     (c)forall(extensionClass(...), newAspect(...)),
     (d)forall(extensionInterface(...), copyExtensionInterface(...)),
     (e)forall(transition(...), createTransition(...)),
        [...]
```

(a) Copy the standard kernel elements
(b) Model variation methods using after aspects and also converts variation method states
(c) Model extension classes using new aspects and also converts extension class states
(d) Copy the extension interfaces
(e) Copy instantiation diagram transitions

Figure 6.9. AOP implementation model

6.1.4 SUBJECT-ORIENTED PROGRAMMING

Subject-oriented programming (SOP) [38] is a paradigm that allows the decomposition of a system into various subjects. A subject compiler [49] can then be used to generate an application by composing the desired subjects. The composition is specified through composition rules such as *Merge* and *Override* [49].

The application of SOP implementation techniques to frameworks leads to an implementation very similar to the one presented for AspectJ. A subject would be used to represent each different implementation for each framework hot-spot, and another subject would be used to represent the framework kernel. The subjects would then be combined through the use of appropriate composition rules to generate the framework instance. Figure 6.10 illustrates the approach.

The implementation model used for generating subject-oriented frameworks is very similar to the one presented for AspectJ, and the approach is also valid only for static hot-spots, since current subject compilers are based on C++. Another disadvantage is that SOP is still experimental and there are no industrial subject compilers. In both cases (AOP and SOP) the satisfaction of the hot-spot OCL specifications and template diagrams are left in the hands of the developer of the aspects and subjects, respectively.

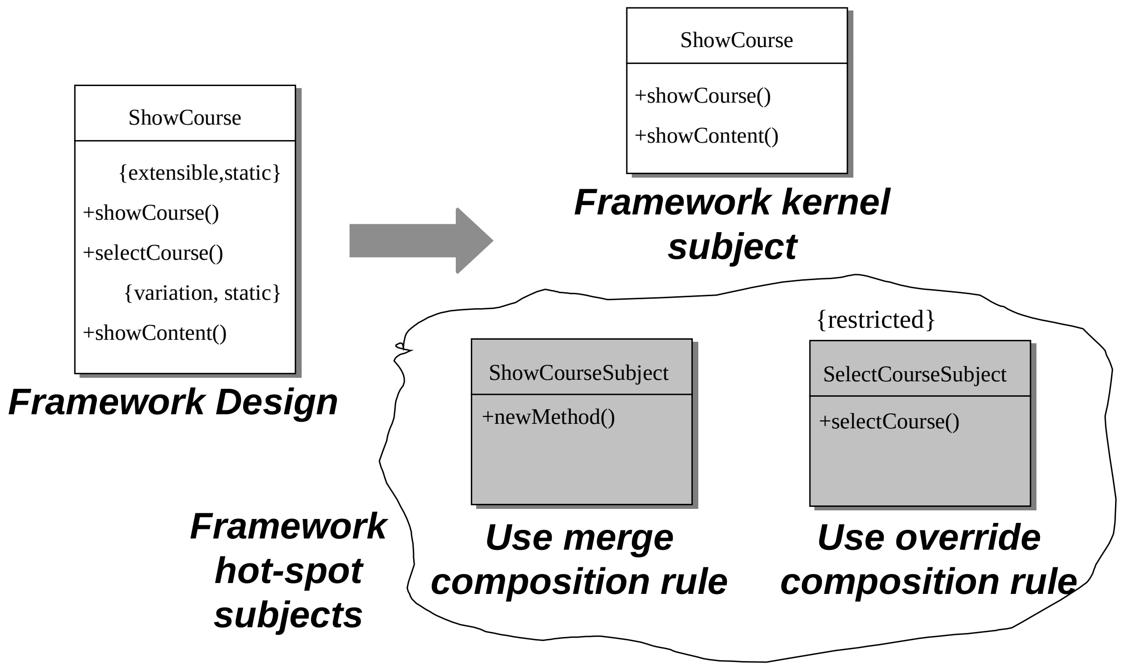


Figure 6.10. Using SOP to implement frameworks

6.2 DOCUMENTATION MODELS

The combination of the framework design representation with an implementation model provides good documentation to assist with framework instantiation. This section further explores this topic by showing how the proposed design language can help the generation of framework documentation to assist the job of the application developer. It describes some documentation techniques and shows how they can be derived from UML-extended design diagrams.

6.2.1 COOKBOOKS AND PATTERNS

Cookbooks [52] provide a textual description of the purpose of the framework, describing its major components and providing examples of its use. However, cookbooks do not have a formal structure, and different cookbooks may focus on different aspects of the framework with different levels of detail. Also the descriptions are quite informal, and this lack of formality may lead to misconceptions by the user. Although the patterns described in [45] are based on an Alexandrian narrative, they respect a pattern form and can be seen as more structured cookbooks. Figure 6.11 presents a pattern for describing the instantiation of the *selectCourse()* hot-spot. It considers that the hot-spot is implemented through the strategy design pattern, as shown in Figure 6.1.

Pattern (1): Defining course selection criteria

(Problem) There can be many possible course selection criteria: courses may or may not require a student login, a list of all available courses may or may not be presented, there is an infinite number of interface layouts (see Changing Layout (9)), and so on.

(Solution and examples) Class **SelectStrategy** is an abstract class which contains an abstract method **select**. Concrete subclasses of **SelectStrategy** that provide concrete implementations for **select** must be provided. Class **ShowCourse** must then invoke the the desired method. See **Strategy** pattern for reference [GoF]

An example is (…)

(Solution Summary) You must define concrete subclasses of **SelectStrategy** and invoke the desired subclass in **ShowCourse.selectCourse**.

(Related patterns) Changing Layout (9) gives a description of how to create different layout representations for the same application object.

Figure 6.11. Pattern for the select course hot-spot

The tool can help the generation of the contents of the *solution* and *related patterns* fields, once an implementation model is chosen. However, the framework developer must support this derivation. The *related patterns* field can be derived, for example, from the template diagrams. In this example the pattern relates this hot-spot with the pattern *Changing Layout*, which is used to describe the *selectionPage()* hot-spot (Figure 3.9(b)). Note that the design diagrams help the documentation process by identifying all the hot-spots that need to be documented.

A lot of information still has to be provided by the framework developer to complete the pattern descriptions, but without an adequate design language this systematic approach would be impossible. A useful enhancement to the pattern form proposed in [45] would be the addition of a new field to represent the hot-spot instantiation restrictions since they are very useful in the instantiation phase.

6.2.2 HOOKS

Hooks [32] can be seen as an enhancement of patterns and cookbooks since hooks describe hot-spots in a more formal way. They classify hot-spots according to their method of adaptation and the amount of support provided by the framework. There are five possible methods of adaptation: enabling a feature, disabling a feature, replacing a feature, augmenting a feature, and adding a feature. Depending upon the amount of support given by the framework, there are three possible classifications: option, supported by pattern, and open-ended.

Enabling a feature allows the execution of a feature that has already been foreseen in the framework design but that is not part of the default execution of the framework. This can be seen as adding a pre-defined component to an extension interface hot-spot. Disabling and replacing a feature allows for variations in the default implementation of the framework and can be modeled as variation method hot-spots. Augmenting a feature allows changes in a feature without disturbing the normal control flow of the framework, and is completely equivalent to our definition of extension class hot-spots. Finally, adding a feature may require the addition and modification of functionality and can be modeled as a combination of extension classes and variation methods hot-spots.

There are different degrees of support provided by the framework. Option adaptations are similar to black-box approaches, in which the framework user only chooses from a set of pre-built components, possibly already defined in a component library. The adaptations supported by patterns are the ones in which there are pre-defined instantiation methods but they are application dependent and have to be filled in by the framework user. Open-ended adaptations have no support and may require complex adaptations of the framework structure.

Each hook description consists of the following parts:

- Name;
- Requirement: the problem it is supposed to solve;
- Type: ordered pair of method of adaptation and level of support;

- Area: parts of the framework affected by the hook;
- Uses: the other hooks required by this one to achieve the complete adaptation of the hot-spot. Similar to the related patterns field in the pattern based description;
- Participants: classes and objects that participate in the instantiation process, including existing classes and the ones that have to be created;
- Changes: formal description using a process language of what changes have to be made and how;
- Constraints: limits imposed by the use of the hook;
- Comments: any additional required information.

Name: Defining course selection criteria

Requirement: There can be many possible course selection criteria: courses may or may not require a student login, a list of all available courses may or may not be presented, there are an infinite number of interface layouts and so on, but only one needs can be used.

Type: Disabling/Replacing a feature, supported by pattern

Area: Course Selection

Participants: SelectStrategy and its newly created subclasses

Changes:

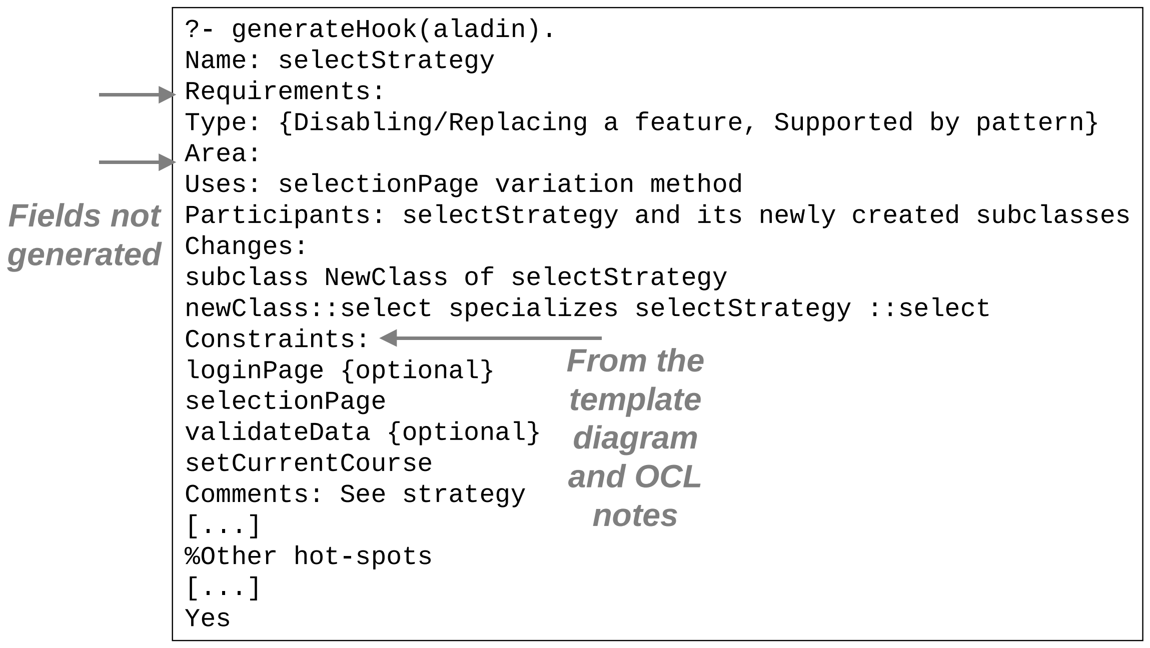
```
subclass ConcreteSelect of SelectStrategy
 ConcreteSelect::select specializes  SelectStrategy::selectCourse
```

Constraints: See the corresponding template diagram;

Comments: Refer to the strategy design pattern [GoF]

Figure 6.12. Example of hook

Figure 6.12 shows an example of a hook for the *selectCourse()* hot-spot. The *changes* field is formally defined using a language and can be automatically generated from the description of the design diagrams, once the desired implementation is selected. The same comment applies to the *type, uses,* and *participants* fields. The hot-spot instantiation restrictions can be directly mapped into the *constraints* field of the hook. The other fields are informal and cannot be supported by a tool, needing to be provided by the framework developer. Figure 6.13 illustrates the generation of hooks by the *documentation tool*.



```
?- generateHook(aladin).
Name: selectStrategy
Requirements:
Type: {Disabling/Replacing a feature, Supported by pattern}
Area:
Uses: selectionPage variation method
Participants: selectStrategy and its newly created subclasses
Changes:
subclass NewClass of selectStrategy
newClass::select specializes selectStrategy ::select
Constraints:
loginPage {optional}
selectionPage
validateData {optional}
setCurrentCourse
Comments: See strategy
[...]
%Other hot-spots
[...]
Yes
```

Figure 6.13. Hook documentation generated by the documentation tool

6.3 RELATED WORK

As discussed in chapter 5, all existing tools that support framework development do not help hot-spot implementation. Since their underlying representation language is based on standard OO constructs there is no way they can automate the selection and application of design patterns, for example. To do so the underlying design language should provide high-level constructs that describe the hot-spot semantics, such as the ones proposed in this work.

Although the design languages supported by the hook tool [33] and the one proposed in [89] explicitly identify the hot-spots, they are also based on standard OO constructs. Consequently, since they do not represent variations and extensions as first-class citizens, the selection of the more appropriate solution to model each framework hot-spot is left in the hands of the framework designer/builder. The implementation models presented in this chapter show how this task can be systematized due to the expression power of the UML-extended framework design language.

Automatic framework documentation can be made if hot-spots are marked in the design. The *changes* field of the hook definitions [32] can be seen as another representation for instantiation diagrams. The same approach for framework documentation described here can be used if frameworks are documented using the language proposed in [89].

Contracts [39, 40] and adaptable plug-and-play components (APPCs) [62] provide linguistic constructs for implementing collaboration-based (or role-based) diagrams in a straightforward manner. They may be used to implement hot-spots since they represent instantiation as first-class citizens. However, these concepts are still quite new and their use for implementing frameworks needs further investigation. Also Lieberherr and the researchers of the Demeter Project [56] have developed a set of concepts and tools to help and evaluate object-oriented design that can be used to enhance framework development.

6.4 SUMMARY

The implementation phase is responsible for generating the framework and the corresponding documentation from the framework design specification. The framework is generated by mapping all the variation methods and extension class hot-spots into existing implementation level constructs. Several approaches may be used to perform this mapping. The *code generation tool* supports the definition of each one of these approaches in the implementation models meta-artifacts.

The *documentation tool* guides the generation of documentation artifacts to assist the application developer during the framework instantiation. Several documentation techniques may be defined in the documentation models meta-artifacts.

The automation of the framework implementation and documentation process can only be achieved if the underlying framework design language distinguishes the hot-spots from the framework kernel and represents the hot-spot semantics (extensions and variations) as first-class citizens.

Chapter 7 – Framework Instantiation

This chapter highlights the role of the UML-extended design language in the automation of the framework instantiation phase. Two approaches for framework instantiation are proposed in this work. The first one is based on the "execution" of the instantiation diagrams and is supported by the process-based instantiation tool. The second one is based on domain-specific languages (DSLs) and is supported by the DSL generator tool.

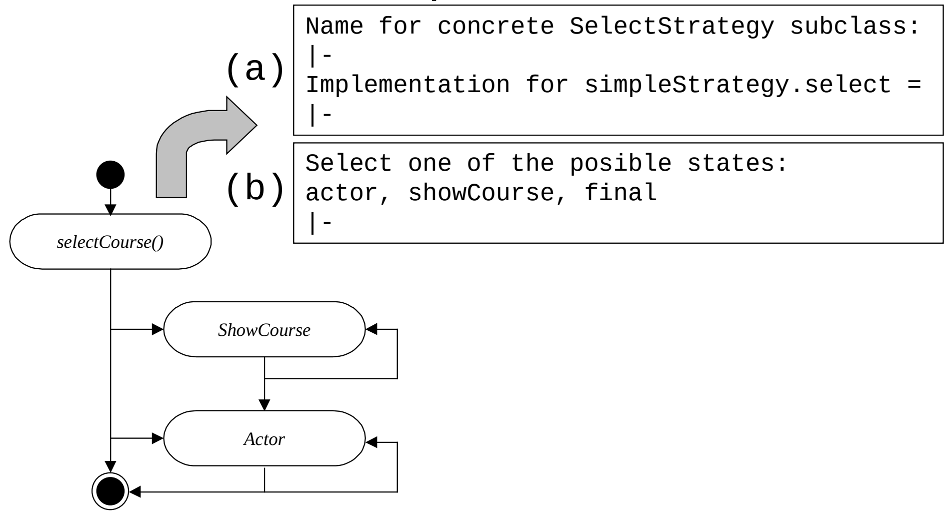
7.1 PROCESS-BASED INSTANTIATION

Framework instantiation can be far more complex than plugging components into framework hot-spots. It is very unlikely that application developers will know how to properly instantiate a real-world framework if the required procedures to do so are not well documented. The UML-extended design language proposes instantiation diagrams as a way of precisely documenting how frameworks should be instantiated.

General procedure

```
For each state {
       (a) Instantiate hot-spot
       (b) List next available states
       }
```

Example



```
Name for concrete SelectStrategy subclass:
|-
Implementation for simpleStrategy.select =
|-
```

```
Select one of the posible states:
actor, showCourse, final
|-
```

Figure 7.1. Process-based instantiation tool algorithm

The *process-based instantiation tool* uses this information to guide the application developer throughout the instantiation phase. It knows how the hot-spots have been implemented and what are the exact procedures needed to instantiate them. The only hot-spots that are instantiated by the tool are the extension interface hot-spots, since the others have already been eliminated during the implementation phase.

Figure 7.1 illustrates how the tool works by describing the actions performed in each state of the interaction diagram being visited. Verification of OCL specifications and template diagrams are not supported in the current implementation of the tool (note that these verifications are generally undecidable).

A more elaborated implementation of this tool should provide support for multi-user development and project management control. The incorporation of the metrics described in [60] is also a possibility that is being investigated.

7.2 DOMAIN-SPECIFIC LANGUAGES

Another approach for framework instantiation is the use of domain-specific languages (DSLs). Each framework has a corresponding DSL and may be instantiated through DSL programs, which are compiled to generate the framework instantiation code.

One of the major problems of directly using the framework implementation language (Java, for example) to instantiate the framework is that it does not know what are the hot-spots and how they should be instantiated. This has some undesirable consequences: (i) there is no indication of where the application developer should write the instantiation code or what code should be written (ii) the implementation language compiler cannot verify the instantiation restrictions, being unable to provide appropriate error messages.

A DSL that “knows” the framework design structure can solve this problems since its syntax can indicate exactly what code should be written. The DSL compiler can realize semantic checks and generate more appropriate error messages.

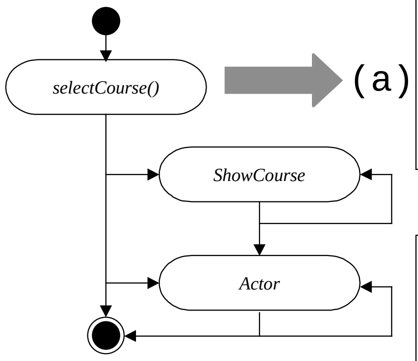
The approach for generating the DSL description is very similar to the one used by the *process-based instantiation tool.* For each state in the framework instantiation diagram the language provides placeholders for each information required to complete the hot-spot definition. The instantiation restrictions derived from the topology of the instantiation diagram are listed as semantic checks that should be verified by the DSL compiler. Figure 7.2 illustrates this generation procedure.

The use of languages other than the framework language to complete the methods missing behavior may enhance the instantiation. The IBM San Francisco is a framework for distributed business applications (http://www.software.ibm.com/ad/sanfrancisco/) that is instantiated through Java programs. However, some of its variation method hot-spots have been provided for defining database queries. Since queries are not a first-class citizen in Java there is no way to provide good error report for the San Francisco application developers if their queries are not well-defined, for example. If a query language is provided, support for the instantiation of those hot-spots is vastly improved. Figure 7.3 illustrates this example.

General procedure

```
For each state {
       (a) Provide placeholders
       }
(b) List intantiation restrictions
```

Example



```
Bloc1 {
Name for concrete SelectStrategy subclass
= %Enter class name here%

Implementation for select =
       { %Enter implementation here% }
}
```

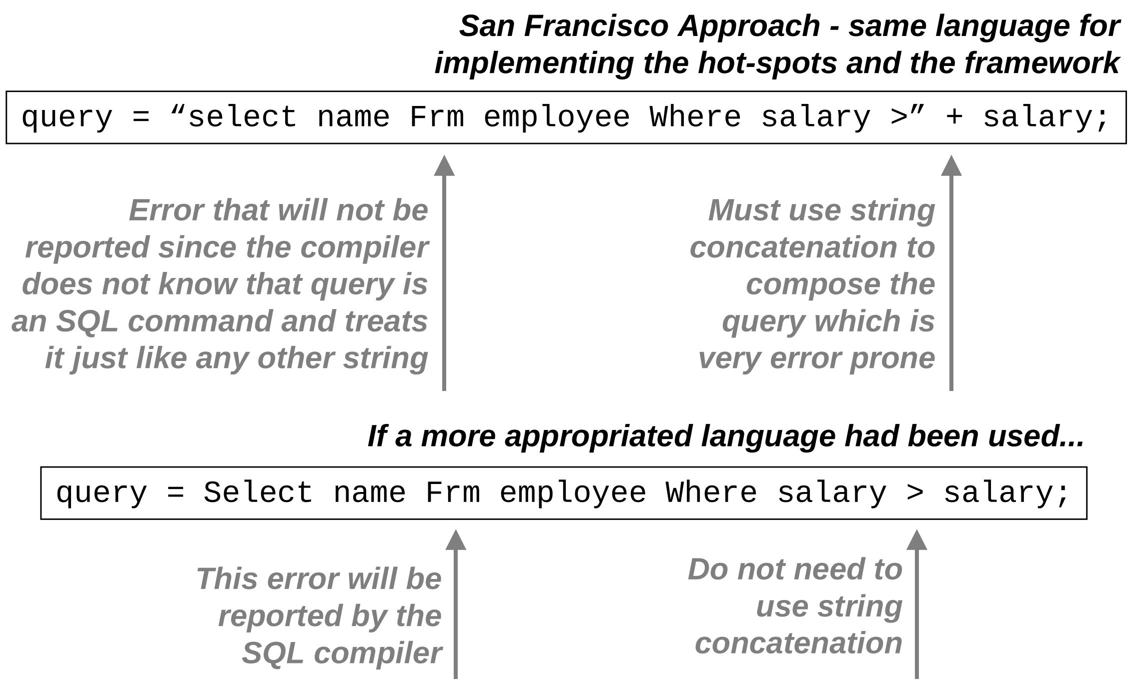
(b)

```
Bloc1 must be defined: 1 time(s)
Bloc2 must be defined: 0 or more time(s)
Bloc3 must be defined: 0 or more time(s)
Bloc2 implies Bloc3
```

Figure 7.2. DSL generator algorithm

However, the definition of the most appropriate language to support the instantiation of each hot-spot is a creative task that cannot be automated. The approach used by the *DSL generator* tool is independent of the language used to implement each hot-spot. It only describes the hot-spot missing information that should be instantiated and their interdependencies. The language generated by the *DSL generator* can be defined as the *framework structure language*, while the languages that are used to implement the hot-spots are the *hot-spot languages* (Figure 7.4). This structure is similar to the one proposed in AOP [50].

The hook tool [33] is in fact a process-based tool, which enacts the hook definitions just as the *process-based instantiation tool* does with the framework instantiation diagrams. Several design pattern tools [12, 24, 29, 61] generate code from the pattern definitions. The approach used in [61] is specially interesting since it explicitly provides support for design pattern instantiation, describing and executing the instantiation procedures for each supported pattern.


San Francisco Approach - same language for implementing the hot-spots and the framework
Error that will not be reported since the compiler does not know that query is an SQL command and treats it just like any other string
Must use string concatenation to compose the query which is very error prone
If a more appropriated language had been used...
This error will be reported by the SQL compiler
Do not need to use string concatenation

```
query = “select name Frm employee Where salary >” + salary;
```

```
query = Select name Frm employee Where salary > salary;
```

Figure 7.3. Using different languages to implement hot-spots

The wizards in existing integrated development environments [10, 63] can be seen as a primitive kind of instantiation tool. In these environments the user interacts with a sequence of dialog boxes in order to generate code. The main drawback is that all generators are hard-wired in these systems, and it is not possible to define new abstractions.

HP Labs has a successful reuse program [36] based on domain-specific kits. Domain-specific kits [37] are designed to help the development of families of related applications and achieve software reuse. A domain-specific kit is the combination of an application framework, a set of reusable components, and glue languages, which may be DSLs and/or wide-spectrum languages such as Java or Smalltalk. The glue languages can be used to connect the components to the framework hot-spots or to generate the missing hot-spot code. The kits may also be composed of other tools (e.g. builders and wizards), documentation, and examples.

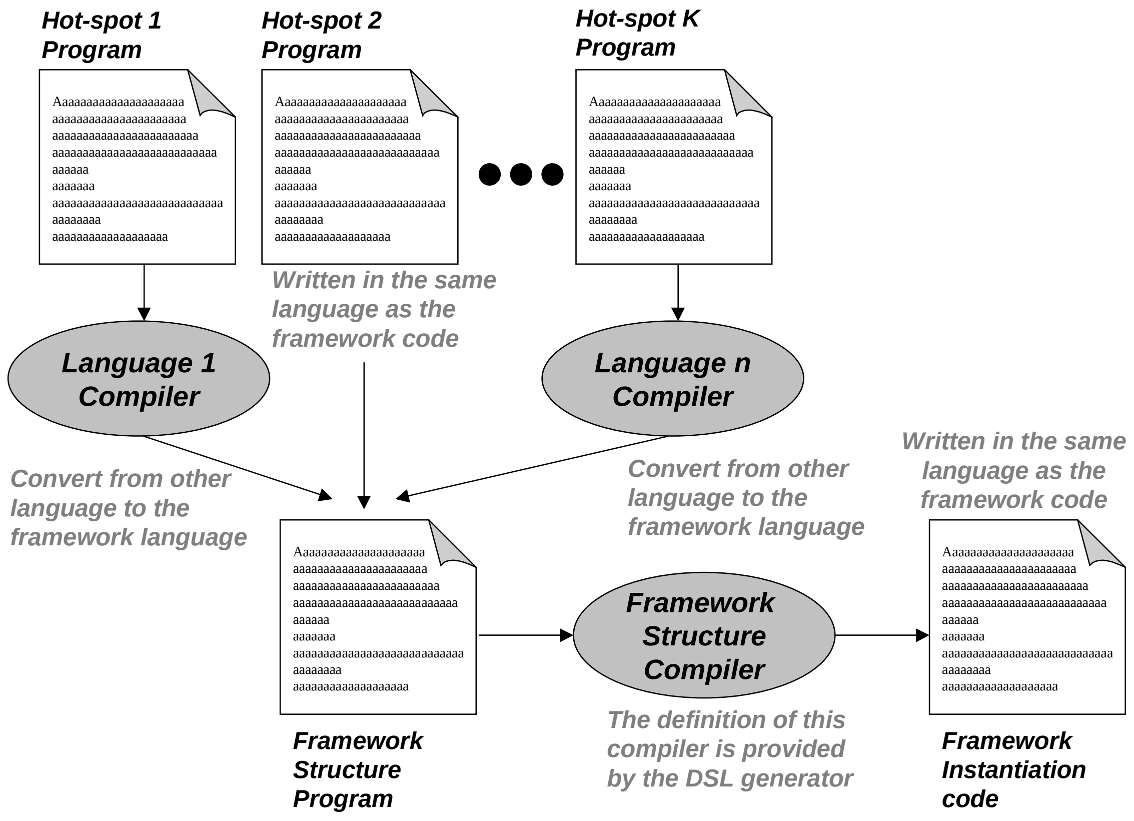

Hot-spot 1 Program
Hot-spot 2 Program
Hot-spot K Program
Written in the same language as the framework code
Language 1 Compiler
Language n Compiler
Convert from other language to the framework language
Convert from other language to the framework language
Written in the same language as the framework code
Framework Structure Compiler
The definition of this compiler is provided by the DSL generator
Framework Structure Program
Framework Instantiation code

Figure 7.4. Combining several DSLs to instantiate frameworks

The work on domain-specific kits can be seen as a practical version of the two approaches for framework instantiation proposed here. Hybrid domain-specific kits are kits that combine the two solutions [37]. However, there is no indication in their work of how the kits are designed and what are the builders/DSLs being used to support framework instantiation.

Several other authors propose the use of DSL to instantiate frameworks [11, 79] but none of them propose an approach for the DSL generation based on the framework structure.

A work for enhancing the utilization of software libraries is described in [73, 93]. The motivation for their work is that end users often do not make effective use of libraries, because most of them lack the expertise to properly identify and compose the routines appropriate to their application. They argue that in domains with mature subroutine libraries, one can greatly improve the productivity and quality of software engineering by automating the effective use of those libraries. The DSL generator approach can be seen as an enhancement of this approach for frameworks. A main difference has to do with the technique used to generate the application. In [73, 93] the code is generated based on a deductive approach [58] using constructive theorem proving. In our technique the code is generated using DSL compilers or transformational systems with generative component concepts [85].

7.4 SUMMARY

During the instantiation phase the application developer must complete the hot-spots' missing information to create applications based on the framework. The proposed framework design language provides an explicit representation of the framework hot-spots and an indication of how they should be instantiated, thus allowing for the construction of supporting tools that automate this development task.

Two different tools are proposed to assist the job of the application developer: the *process-based instantiation tool* and the *DSL generator*. The *process-based instantiation tool* provides a compositional approach to framework instantiation [37], in which a process description guides the way components are integrated into the framework. The *DSL generator,* on the other hand, provides a generative approach to framework instantiation [6, 37], in which the framework instantiation is based on DSL programs that are then compiled to complete the missing hot-spot information. *DSL generator* generates the definition of the *framework structure language*. If languages other than the one used to implement the framework are required *(hot-spot languages)* they have to be defined without tool support. The process of choosing the most appropriate language to instantiate each hot-spot is creative, and cannot be automated.

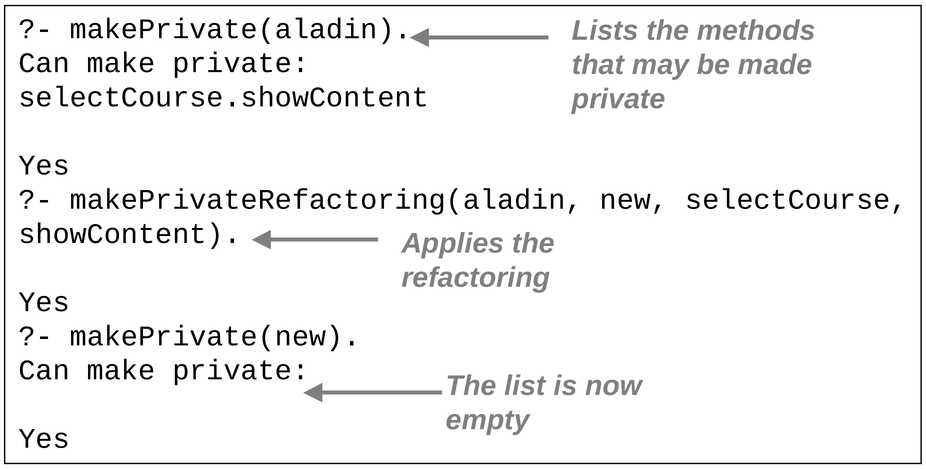
Chapter 8 – Framework Maintenance

This chapter describes the use of refactoring and unification rules to support framework maintenance. Tool support for both these transformations, provided by the maintenance tool, is also presented. Finally, a software development process based on unification rules is proposed for developing frameworks for "changing" or non-established domains.

8.1 REFACTORING RULES

Refactoring rules are already an established concept [47, 66, 78] and are independent from the design language proposed in this work. They are useful for restructuring frameworks based on design heuristics [76], for example, minimizing a class interface. The *maintenance tool* provides analyses for checking when refactoring procedures should be applied and transformations that effectively realize the refactorings.

Figure 8.1 illustrates the use of a design analysis that checks what methods of a project could become *private* to minimize the class interfaces. After verification the actual refactoring is applied.



```
?- makePrivate(aladin).
Can make private:
selectCourse.showContent

Yes
?- makePrivateRefactoring(aladin, new, selectCourse,
showContent).

Yes
?- makePrivate(new).
Can make private:

Yes
```

Figure 8.1. Checking and applying refactorings

The refactoring analyses are very similar to the design analyses described in chapter 5. They use the knowledge base information about the software designs and cannot modify it. When the refactorings are applied by the *maintenance tool* a new project is created to store the design generated by the transformation. In the example in Figure 8.1, the project *aladin* is transformed into the project *new* when refactoring takes place.

8.2 UNIFICATION RULES

Framework instantiation normally takes place by completing the hot-spots in the framework. However, there are many cases in which the framework does not support the required customization and the application developers need to violate its structure. This phenomenon is referred to as architectural drift: the intended framework architecture and the architecture that underlies the current implementation of a given framework instance become different [16]. Unification rules can be used to avoid this phenomenon, by making the necessary transformations in the framework structure to incorporate the changes required by a given framework instance. The term *unification* is used to indicate that the rules are used to "unify" the framework architecture with the architecture of the violating instance.

Currently there are three unification rules:

- Variation method unification: is applied when an instance must change the implementation of a kernel method. After this transformation is applied the method is defined to be a variation method hot-spot. This unification has already been illustrated in Figure 4.16;
- Extension class unification: is applied when an instance must add new methods to a kernel class. After this transformation is applied the class is defined to be an extension class hot-spot. Figure 8.2 illustrates the use of this rule;
- Extension interface unification: is applied when new class must be added to the framework, and this addition is not supported by any of the framework extension interfaces. After this transformation a new extension interface is created. It can be a new class/interface or it can be one of the classes/interfaces already defined in the framework structure. One example of the use of this rule is presented in Figure 8.3.

The application of unification transformations has to be assisted by a human actor since several parameters need to be provided, such as the hot-spot run-time requirements (if they are static or dynamic), their instantiation restrictions (OCL notes and template diagrams), and their participation in the framework instantiation diagram.

Unlike refactorings, unification rules may only be supported if the framework is represented through a design language that explicitly represents its hot-spots.

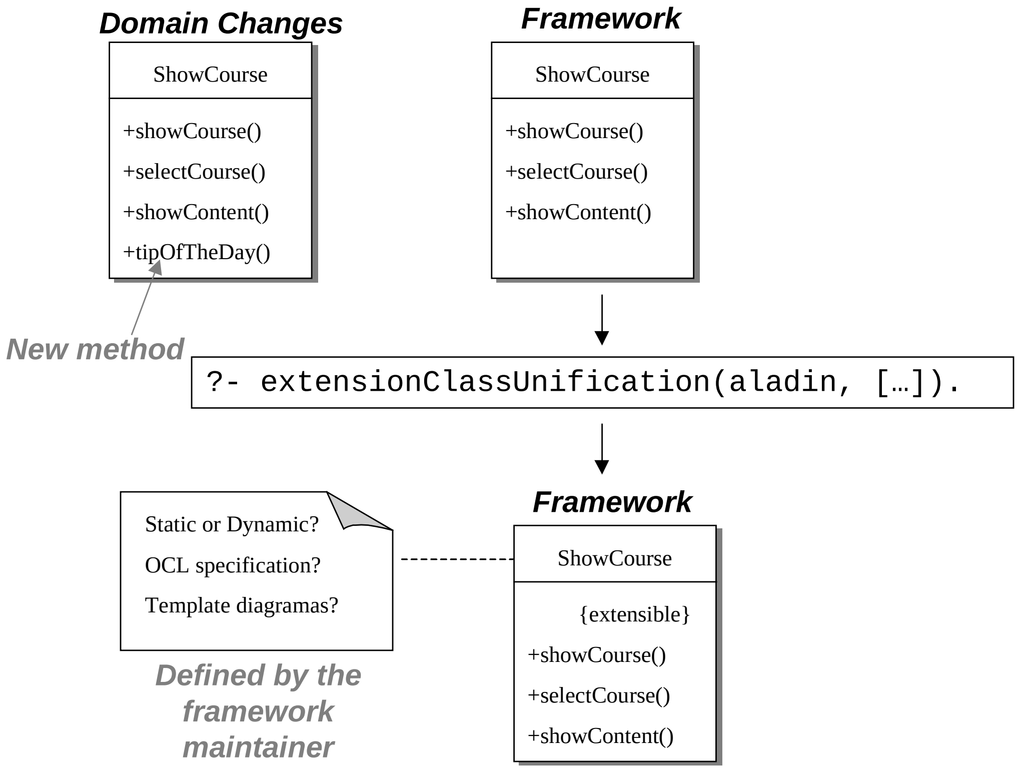


Figure 8.2. Extension class unification example

When unification rules are applied, the maintainer tool creates a new copy of the project, as in the case of refactorings.

8.3 UNIFICATION-BASED DEVELOPMENT PROCESS

There are various application areas that are not yet established and for which new ideas and models are presently under development and evaluation. These are domains in which the possibility of rapidly building new applications is essential and strategic from a practical point of view. Examples of application domains that can be classified in this category include web-based education, electronic commerce, computational biology, and finances.

It is very difficult to develop an adequate framework for these domains without a lot of experimentation. One possible solution for automating part of the job is the use of a unification-based development process.

The idea is to develop a first approximation of the framework and start to develop applications from it. Some of these applications will violate the framework architecture. Every time this happens, the unification rules may be applied generating a more mature version of the framework. When no more unifications are required the framework can be considered sufficiently mature. Figure 8.4 illustrates this process.

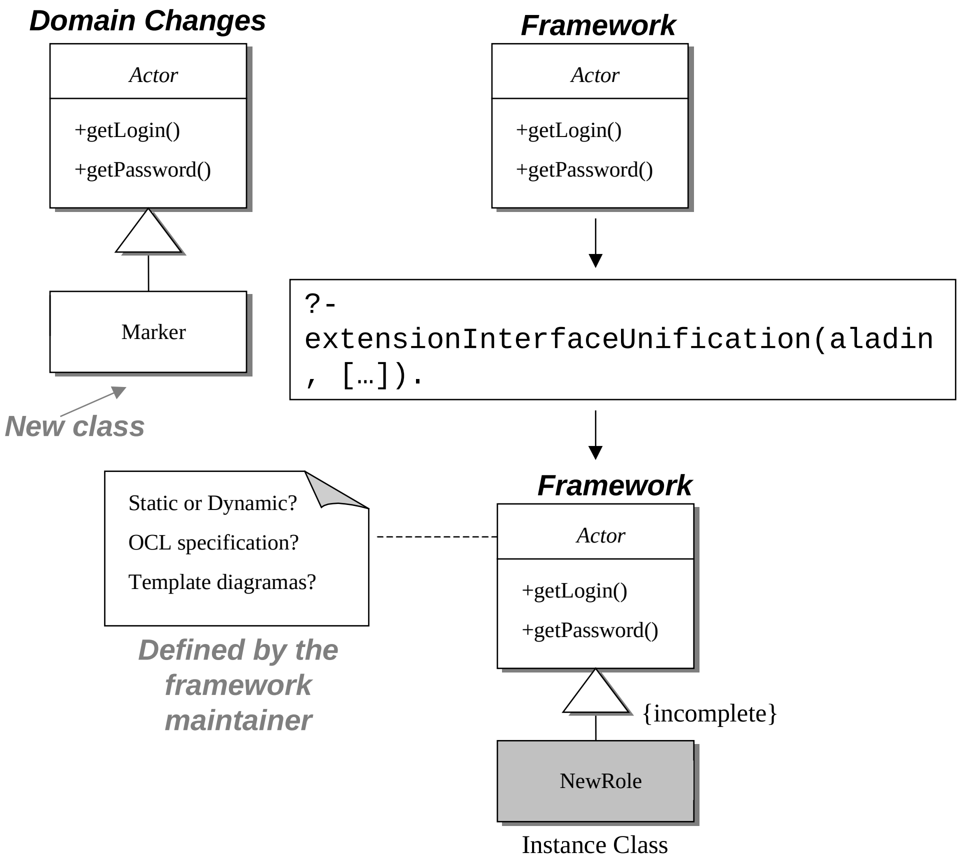


Figure 8.3. Extension interface unification example

The ALADIN framework [31] was defined using this approach, as described in chapter 10 (and also in [2]). The unification transformations illustrated in section 8.2 have been really applied as part of this development process. However, until now this was our only experience with a framework development based on unification rules. In order to really evaluate the merits of the approach new case studies need to be developed.

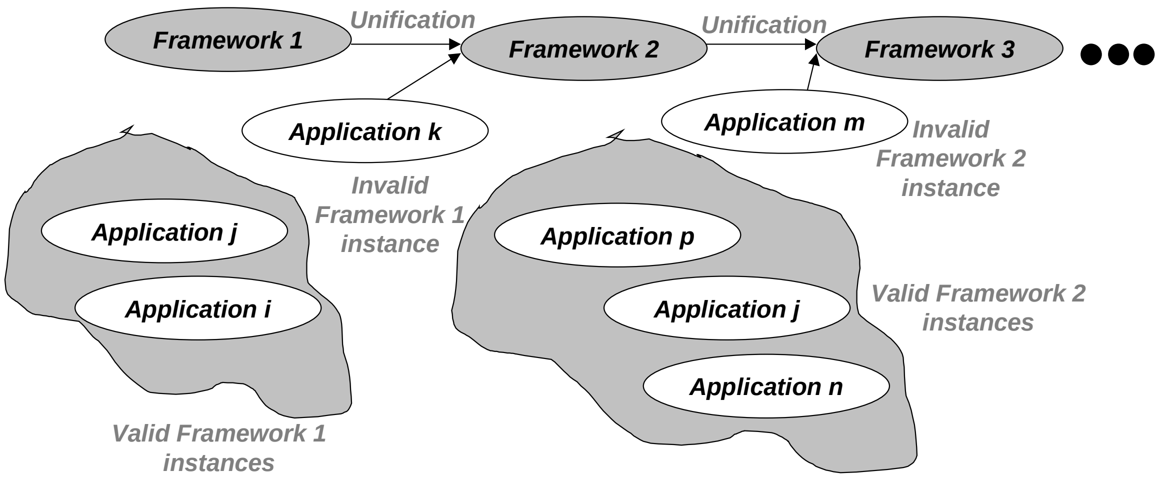


Figure 8.4. Unification-based software development

8.4 RELATED WORK

The Refactoring Browser [78] is a tool to help maintenance of frameworks written in Smalltalk. It currently does not support unification rules, but it has an open architecture and the introduction of unification and new refactoring procedures seems to be straightforward. The design pattern tool proposed in [29] also uses refactorings to achieve framework restructuring. However, none of these tools provide design analyses for checking when refactorings should be applied. Note that there is no way to check when unifications need to be applied, since they are applicable only when framework instances violate the hot-spot structure.

Roberts and Johnson propose the development of concrete applications before actually developing the framework itself [79]. They claim that framework abstractions can be derived from concrete applications. The unification-based development process may be used to systematize this approach.

8.5 SUMMARY

The maintenance tool supports the verification and application of refactorings [47, 66, 78] and the application of unification rules. Refactorings are horizontal behavior-preserving transformations that are used to restructure the framework design (or implementation) based on some design heuristics [76].

Unifications are transformations used to avoid the architectural drift problem [16] by restructuring the framework hot-spots. After the application of a unification transformation the set of applications that may be instantiated based on the framework is enlarged. Unification rules can be used as a basis for a development process that produces frameworks for “changing” domains. This idea can be seen as a systematization of the concrete examples pattern, proposed in [79].

Chapter 9 – Language Semantics

This chapter formalizes the UML-extended design language. The formalization approach is based on set theory and meaning functions. A study of the computational complexity for several semantic checks is presented, illustrating how tools that support these verifications may be constructed. Several implementation, instantiation, and maintenance transformations presented throughout the dissertation are verified based on the language semantics. The verifier tool is presented as an implementation example.

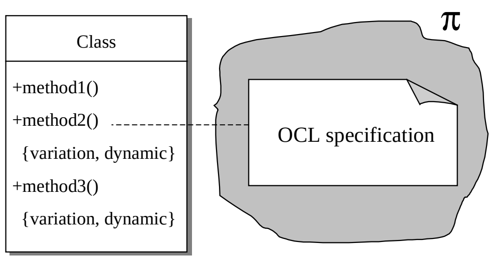
9.1 DESIGN LANGUAGE SEMANTICS

Each design described in the framework design language represents a point in the semantic plan, as shown in Figure 2.6. The *Meaning* function calculates the set of applications that may be instantiated from the design, providing its semantics. The semantics of a design is based on the semantics of its classes, which in turn are based on the semantics of its methods.

The meaning of a method is its implementation. If Φ is the set of all Turing-computable functions, the meaning of a kernel method is f, such that $f \in \Phi$. In the case of variation methods, it is the set of all possible implementations. Variation hot-spots that have no instantiation restrictions (OCL notes or template diagrams) have the entire set Φ as its semantics. If instantiation restrictions are provided, the semantics of variation method hot-spots are given by sets $H_i = \{h \mid \Pi_i\}$, such that $H_i \subseteq \Phi$ and each Π_i is a set of properties that describe the restrictions that must be followed by the hot-spot i. Thus, each implementation h that belongs to H_i satisfies Π_i. Figure 9.1 illustrates these three possibilities. The class attributes define the "environment" of the methods limiting their behavior, however environments will not be part of the method semantics since they are irrelevant for the purposes of this work. If necessary they can be easily added later on.

The meaning of a class is the combination of the meanings of all of their methods. If the class is isolated (not related to any other class in the system) its semantics is provided only by its own methods. The meaning of the isolated class *Class* in Figure 9.1 is the set generated by all the combinations of the product $f \times H \times \Phi$, as illustrated in Figure 9.2.

The product $f \times H \times \Phi$ is the *symbolic description* of the class semantics.



Meaning(method1) = f
Meaning(method2) = $H = \{h \mid \Pi\}$
Meaning(method3) = Φ

where: $f \in \Phi$
$H \subseteq \Phi$

Figure 9.1. Method semantics

Meaning(Class) = $\{ <f, h_1, f_1>, <f, h_1, f_2>, ..., <f, h_n, f_k>, ...\}$
where:
Meaning(method1) = f
Meaning(method2) = $H = \{h \mid \Pi\}$
Meaning(method3) = Φ
$f \in \Phi$
$H \subseteq \Phi$
$\forall i,j \bullet h_i \in H \wedge f_j \in \Phi$

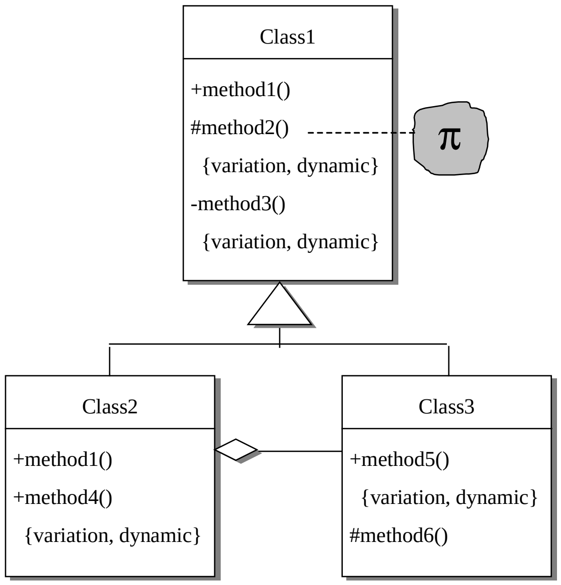
Figure 9.2. Meaning of an isolated class

For classes that belong to inheritance hierarchies the methods defined in their super classes have to be considered, unless they are private methods. This situation is illustrated in Figure 9.3. In this example the class meanings are represented using their symbolic descriptions instead of enumerating the sets. The use of the class name before the method semantics identifies which class the methods came from. In this example, *Class2* and *Class3* inherit methods *method1*, which is public, and *method2*, which is protected, from their common super class *Class1*. Method *method3* is not inherited since it is a private method. Note that method *method1* was overridden in *Class2*, and for this reason it may have a different implementation and consequently a different semantics. UML uses + , #, and – to represent the public, protected, and private access status, respectively.

If none of the methods of a class is a variation method, its symbolic description has the form $f_i \; x \; f_j \; x \; ... \; x \; f_n$. The cardinality of the set generated by symbolic descriptions of this kind is always one *(#Meaning(class) = 1).*

The semantics of a design is the combination of the semantics of all of its classes. In the case of Figure 9.3, the meaning of the design has the symbolic representation *Meaning(Class1) x Meaning(Class2) x Meaning(Class3).*

Extension classes are classes that contain possibly infinite methods, since its interface may be extended during framework instantiation. If instantiation restrictions are provided they restrict the set of methods that may be added to the class. The procedure to calculate the semantics of extension classes is very similar to the one used for standard classes: the only difference is the use of the repetition symbol * to designate the set of new methods that might be added to it. Figure 9.4 illustrates the semantics of extension classes when instantiation restrictions are provided (a) and when they are not provided (b).

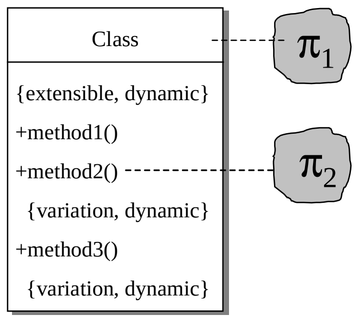


Meaning(Class1) ≈ Class1. f_1 x Class1.H x Class1.Φ
Meaning(Class2) ≈ Class2. f_1' x Class1.H x Class2.Φ
Meaning(Class3) ≈ Class1. f_1 x Class1.H x Class3. Φ x Class3. f_6

where:
Meaning(Class1.method1) = Class1.f_1
Meaning(Class1.method2) = Class1.H
Meaning(Class1.method3) = Class1.Φ
Meaning(Class2.method1) = Class2. f_1'
Meaning(Class2.method4) = Class2.Φ
Meaning(Class3.method5) = Class3.Φ
Meaning(Class3.method6) = Class3. f_6
$f_1, f_6 \in \Phi$
$H_2 \subseteq \Phi$

Figure 9.3. Meaning of non-isolated classes

Interfaces, abstract classes, and abstract methods have no semantics since they are only used to organize the design and may be eliminated at run-time. However, extension interfaces allow the definition of new classes in the system during the framework instantiation, which are the instance classes, and the semantics of these new classes have to be taken in account when calculating the semantics of the design. The set of new classes that might be added is represented through the repetition symbol *. Figure 9.5 illustrates how the semantics of a design that contains only one extension interface is calculated, with (a) and without instantiation restrictions (b).



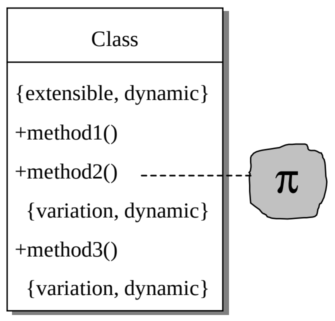
For other methods that might be added

$$\text{Meaning(Class)} \approx f_1 \text{ x } H_2 \text{ x } \Phi \text{ x } H_1^*$$

where:

$$\text{Meaning(method1)} = f_1$$
$$\text{Meaning(method2)} = H_2 = \{h \mid \Pi_2\}$$
$$\text{Meaning(method3)} = \Phi$$
$$f_1 \in \Phi$$
$$H_2 \subseteq \Phi$$
$$H_1 = \{h \mid \Pi_1\} \wedge H_1 \subseteq \Phi$$

(a)



For other methods that might be added

$$\text{Meaning(Class)} \approx f_1 \text{ x } H \text{ x } \Phi \text{ x } \Phi^*$$

where:

$$\text{Meaning(method1)} = f_1$$
$$\text{Meaning(method2)} = H = \{h \mid \Pi\}$$
$$\text{Meaning(method3)} = \Phi$$
$$f_1 \in \Phi$$
$$H \subseteq \Phi$$

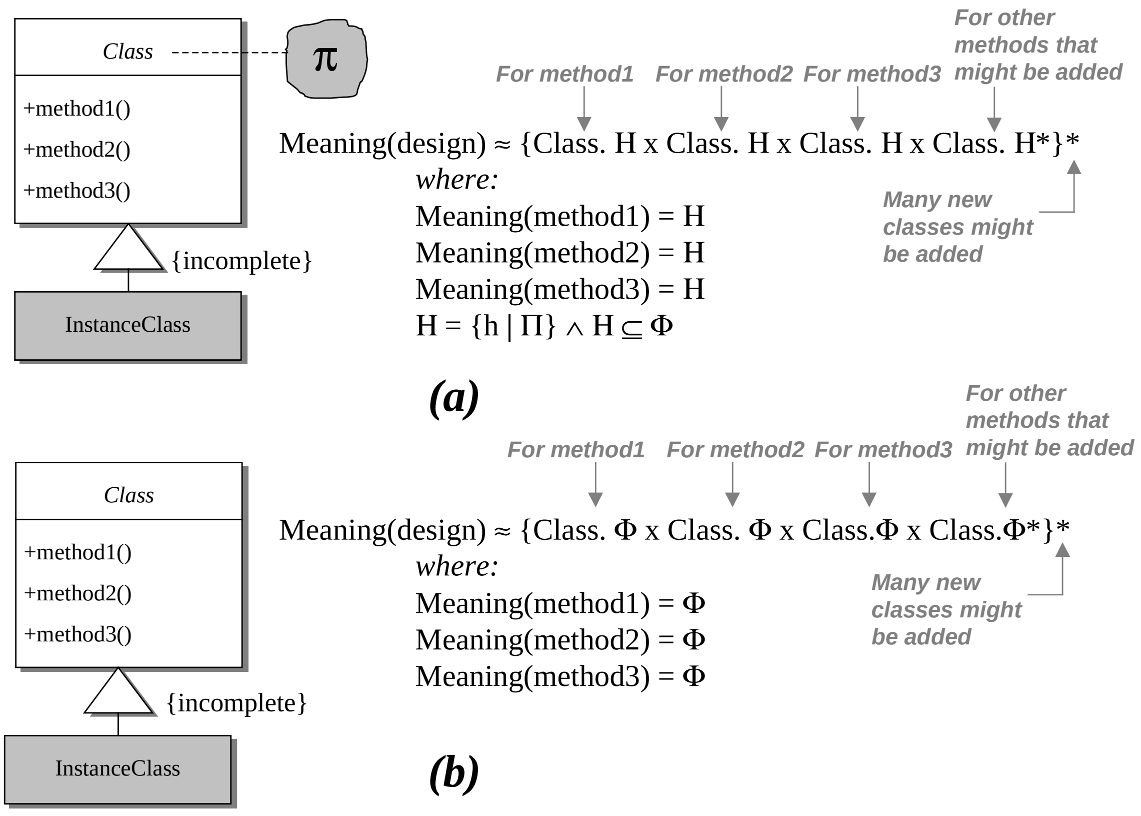
(b)

Figure 9.4. Extension class semantics

Figure 9.6 shows the semantics of a design of an instance application that was defined from the framework modeled in Figure 9.5. In this example, the first class instance did override the three methods but did not add any other method. The second class only overrode *method1*, and added a new method. Finally, the third class overrode *method2* and *method3* and added two new methods. Method overriding is indicated by designating another symbol to represent the method semantics. In this example apostrophes were used for this purpose.

Restricted extension interfaces do not allow the addition of new methods. Their semantics are calculated in a way similar to the one used for the semantics for standard extension interfaces, as illustrated in Figure 9.7. It should be clear that the instance classes derived from standard extension interfaces are extension classes, while the ones derived form restricted extension interfaces are not.

If a design has no variation methods, extension classes, or extension interfaces the cardinality of its meaning is one. On the other hand, if it has at least one hot-spot then *#Meaning(design) >1* and the system modeled by *design* is a framework.



$$Meaning(design) \approx \{Class.\ H \times Class.\ H \times Class.\ H \times Class.\ H^*\}^*$$

where:

$$Meaning(method1) = H$$
$$Meaning(method2) = H$$
$$Meaning(method3) = H$$
$$H = \{h \mid \Pi\} \wedge H \subseteq \Phi$$

$$Meaning(design) \approx \{Class.\ \Phi \times Class.\ \Phi \times Class.\Phi \times Class.\Phi^*\}^*$$

where:

$$Meaning(method1) = \Phi$$
$$Meaning(method2) = \Phi$$
$$Meaning(method3) = \Phi$$

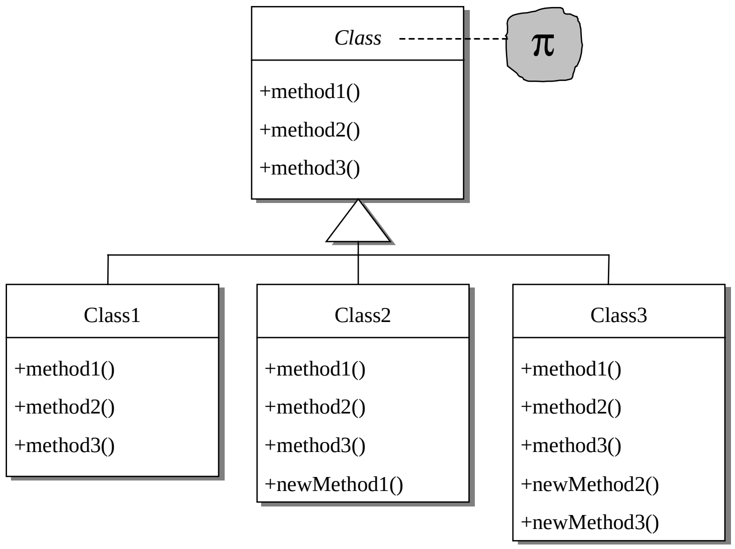
Figure 9.5. Meaning of extension interfaces

9.2 COMPUTATIONAL COMPLEXITY

Although the set that represents the semantics of a framework may be infinite, its symbolic representation may be calculated in polynomial time.

The semantics of each method is calculated in *O(1)*. The algorithm assigns a different symbol f_i to represent the semantics of each different kernel method in the system. The semantics of variation methods that have no instantiation restrictions is always Φ. Finally, the semantics of variation methods that have instantiation restrictions is given by a set H_i, where $H_i = \{h \mid \Pi_i\}$. Thus, a different symbol H_i is assigned to each different hot-spot that has instantiation restrictions.

The semantics of a class that has *M* methods is calculated in *O(M)*. The same is true for extension classes, since their semantics is calculated with a procedure similar to the one used to calculate standard classes. The only difference is that the semantics of extension classes is complemented with H_i* or $\Phi*$, depending on whether it has to follow instantiation restrictions or not.



$$\begin{aligned} \text{Meaning(instance)} \approx\ & \{\text{Class.f}_1\text{'} \times \text{Class.f}_2\text{'} \times \text{Class. f}_3\text{'}\} \times \\ & \{\text{Class.f}_1\text{''} \times \text{Class.f}_2 \times \text{Class. f}_3 \times \text{Class. h}_1\} \times \\ & \{\text{Class.f}_1 \times \text{Class.f}_2\text{''} \times \text{Class. f}_3\text{''} \times \text{Class. h}_2 \times \text{Class.h}_3\} \\ & \textit{where:} \\ & H = \{h \mid \Pi\} \wedge H \subseteq \Phi \\ & \forall i,j \bullet h_i \in H \wedge f_j \in \Phi. \end{aligned}$$

Figure 9.6. Instantiation example

The semantics of a design that has *C* classes and *M* methods can be calculated in *C*O(M)* or *O(C*M)* in the worst case, which is when every class in the system contains every method. If the design has extension interfaces this complexity is still the same since they are calculated in the same way as extension classes. The only difference is that their representation is marked with an extra repetition symbol *. Restricted extension interfaces are calculated just as standard classes and marked with the repetition symbol.

The verification of behavior-preserving transformations is generally an undecidable task. However, a tool that assists this task in polynomial time may be defined. Let C_b, M_b, C_a, and M_a be the number of classes and methods of the designs *before* and *after* the transformation, respectively. The verification of the semantics of the transformation is performed by calculating *Meaning(before)*, which takes $O(C_b*M_b)$, *Meaning(after)*, which takes $O(C_a*M_a)$, and checking if they are equal. Since this last test is undecidable, the tool cannot automatically calculate it. What it can do is to eliminate the elements that exist in both sides of the equality expression and show the simplified expression to the user. This simplification is performed in $O(C_a*M_a) + O(C_b*M_b)$. Thus, the total time for verifying the transformation takes $2*O(C_b*M_b) + 2*O(C_a*M_a)$, which may be written as $O(C_b*M_b+C_a*M_a)$. Sections 9.3 and 9.5 present some examples of the verification of behavior-preserving transformations applied in the implementation and maintenance phase, respectively.

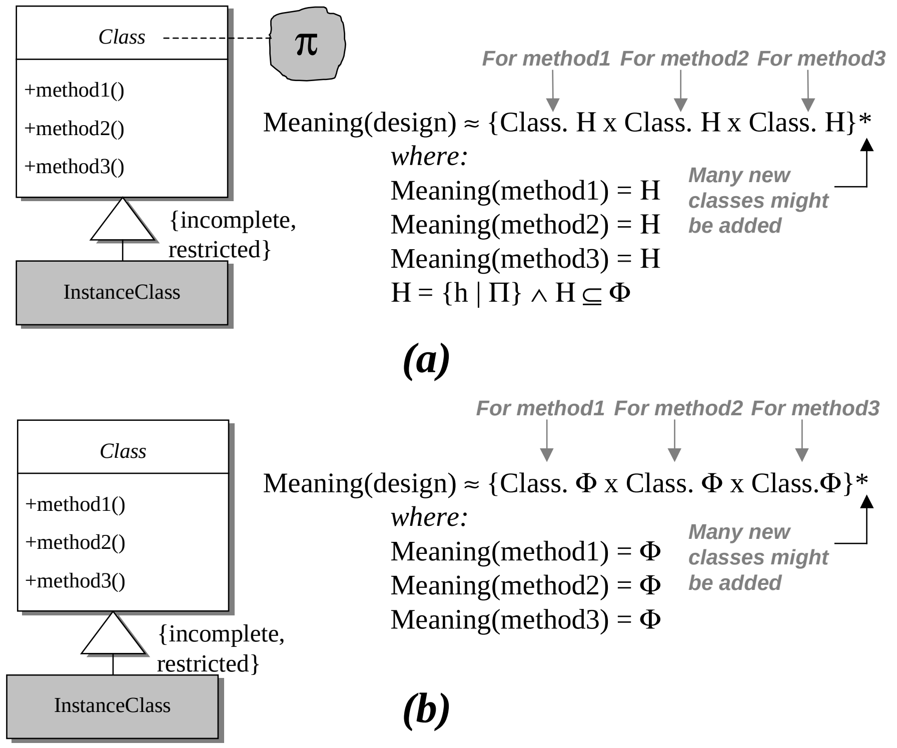


Figure 9.7. Semantics for restricted extension interfaces

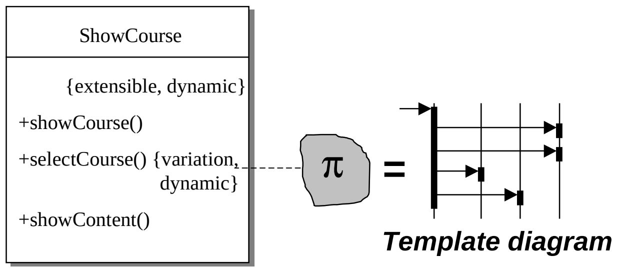
The verification *IsInstance(framework, application)* can be calculated in polynomial time, but only if the design represented by *framework* has no instantiation restrictions. In which case at least one hot-spot has to follow some property Π_i *IsInstance* becomes an undecidable procedure. The reason is that it cannot automatically check if a given implementation h_i belongs to H_i. Algorithms that implement *IsInstance* have to calculate the symbolic representation of the two designs and check if *Meaning(application)* $\subset$ *Meaning(framework).* This is done by verifying if each element in the symbolic representation of *Meaning(application)* is a subset of its corresponding element in the symbolic representation of *Meaning(framework)*. If C_a, M_a are the number of classes and methods in the *application* this verification is performed in $O(C_a*M_a)$. Thus, the total time to calculate *IsInstance* is $2*O(C_a*M_a) + O(C_f*M_f)$, which may be written as $O(C_b*M_b+C_a*M_a)$, where C_f, M_f are the number of classes and methods in *framework.* Note that for valid instances it is always true that $C_a > C_f$ and $M_a > M_f$, since new methods and classes may be added to *application* during the instantiation phase. Section 9.4 illustrates how *IsInstance* is calculated to verify instantiation transformations.

9.3 IMPLEMENTATION TRANSFORMATIONS

Figure 9.8 illustrates part of the web-based education framework [31] and the symbolic representation of its design[7]. This section will use this framework as an example to illustrate how several transformations presented in the implementation models described in chapter 6 may be verified.

Implementation transformations are responsible for eliminating the variation methods and extension classes of a design, replacing them by appropriate implementation level constructs. These transformations have to be behavior-preserving, since the implementation approach chosen for the framework hot-spots must not change the semantics of the framework.

[7] This is a variation of the design presented in Figure 3.8 that does not consider instantiation restrictions for the *ShowCourse* class hot-spot and the *Actor* hierarchy



$$\text{Meaning(design)} \approx \{SC.f_1 \text{ x } SC.H \text{ x } SC.f_2 \text{ x } SC.\Phi^*\}$$

where:

Meaning(ShowCourse.showCourse) = $SC.\ f_1$

Meaning(ShowCourse.selectCourse) = $SC.\ H$

Meaning(ShowCourse.showContent) = $SC.\ f_2$

$H = \{h \mid \Pi\} \wedge H \subseteq \Phi$

Figure 9.8. The web-based education framework and its semantics

9.3.1 DESIGN PATTERN TRANSFORMATIONS

Figure 9.9 illustrates a framework generated from Figure 9.8 design representation, using the implementation approach based on the strategy and visitor patterns [34] to model variation classes and extension method hot-spots, respectively. It also presents the symbolic representation of the semantics of the generated framework.

To prove that the transformation is behavior-preserving the system must prove that

$$\{SC.f_1 \text{ x } SC.H \text{ x } SC.f_2 \text{ x } SC.\ \Phi^*\} = \{SC.f_1 \text{ x } SC.f_3 \text{ x } SC.f_2 \text{ x } SC.\ f_4\} \text{ x } \{SS.H\}^* \text{ x } \{V.\Phi\}^*$$

The elements that exist on both sides may be directly eliminated. Those are the elements that did not participate in the transformation. Thus, the expression may be simplified to

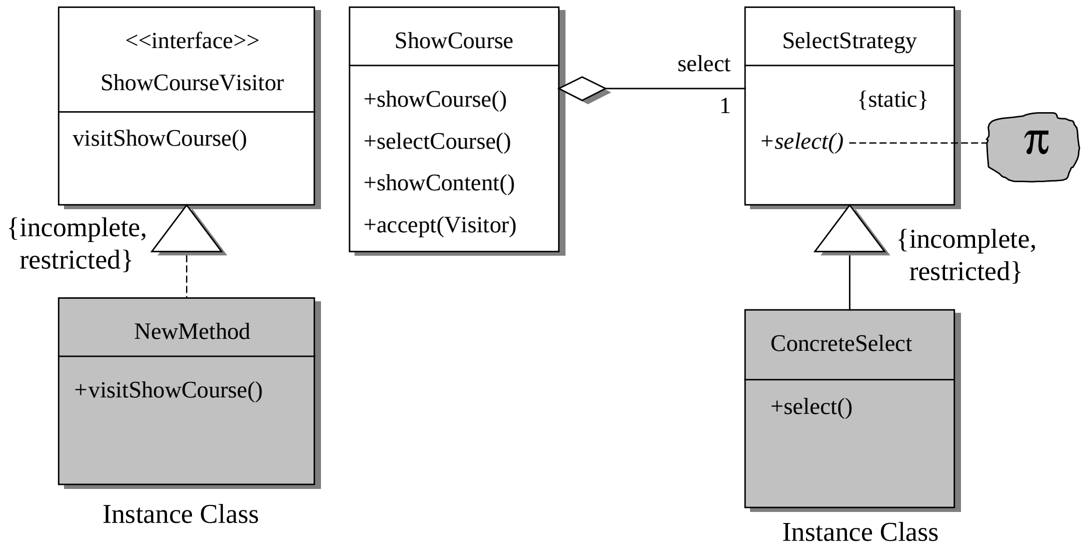
$$\{SC.H \text{ x } SC.\Phi^*\} = \{SC.f_3 \text{ x } SC.\ f_4\} \text{ x } \{SS.H\}^* \text{ x } \{V.\Phi\}^*$$

This simplification may be calculated in polynomial time. The rest of the proof has to be assisted by a human actor. However, if the tool knows which transformations have been used, it can guide the user in the proof process. This is the case of the *verifier tool*. In this example there are two transformations that must be proved: (i) the variation method-strategy transformation and (ii) the extension class-visitor transformation, as follows:

$$\text{(i) } \{SC.\ H\} = \{SC.f_3\} \text{ x } \{SS.H\}^* \text{ and (ii) } \{SC.\Phi^*\} = \{SC.\ f_4\} \text{ x } \{V.\Phi\}^*$$

In transformation (i) the method *selectCourse()* uses delegation to invoke method *select()*, in one of the *SelectStrategy* subclasses. Thus, the implementation *{SC.f₃}* just repasses the control to one of the *{SS.H}** instances. Knowing that, expression $\{SC.f_3\}$ *x*

*{SS.H}** may be simplified to *{SC.H}**. Finally, as just one *SelectStrategy* subclass is invoked at a given time the repetition symbol may be eliminated. Thus, the variation method-strategy transformation is proved to be behavior-preserving.



Meaning(design) ≈ $\{SC.f_1 \times SC.f_3 \times SC.f_2 \times SC.f_4\} \times \{SS.H\}^* \times \{V.\Phi\}^*$

where:

Meaning(ShowCourse.showCourse) = $SC.f_1$

Meaning(ShowCourse.selectCourse) = $SC.f_3$

Meaning(ShowCourse.showContent) = $SC.f_2$

Meaning(ShowCourse.accept) = $SC.f_4$

Meaning(SelectStrategy.select) = $SS.H$

Meaning(ShowCourseVisitor.visitShowCourse) = $V.\Phi$

$H = \{h \mid \Pi\} \wedge H \subseteq \Phi$

Figure 9.9. Semantics of design pattern-based implementation

A geometric view of this proof is presented in Figure 9.10. Since *selectCourse()* uses object delegation to invoke the appropriate select *method()*, it calls the same instance class every time (which is the class that models the select object). That is why the invocations are exclusive and the repetition symbol may be eliminated from the hot-spot semantics.

In transformation (ii) the method *accept(Visitor)* invokes the method *visitShowCourse()* in a given *ShowCourseVisitor* subclass. The selection of what subclass to use is defined by the parameter *Visitor*. Thus, every time *accept(Visitor)* is called a new different *visitShowCourse()* may be invoked. The expression *{SC.* f_4*} x {V.Φ}** can then be simplified to *{SC.Φ}**, which is equivalent to *{SC.Φ*}.* This means transformation (ii) is a behavior-preserving transformation.

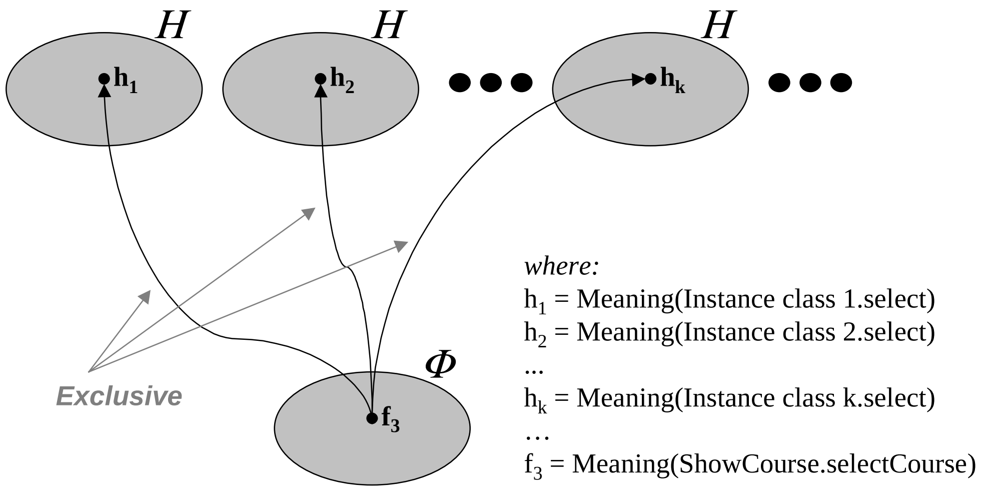

H
H
H
h1
h2
hk
Φ
f3
Exclusive
where:
h1 = Meaning(Instance class 1.select)
h2 = Meaning(Instance class 2.select)
...
hk = Meaning(Instance class k.select)
…
f3 = Meaning(ShowCourse.selectCourse)

Figure 9.10. Geometric view of the variation method-strategy transformation

A geometric view of this proof is presented in Figure 9.11. Since every time *accept(Visitor)* is called a different instance class may be passed as a parameter, many different methods may be used to extend *ShowCourse* interface, each one defined in a different instance class. That is why the invocations are non-exclusive and the repetition symbol is necessary to define the hot-spot semantics.

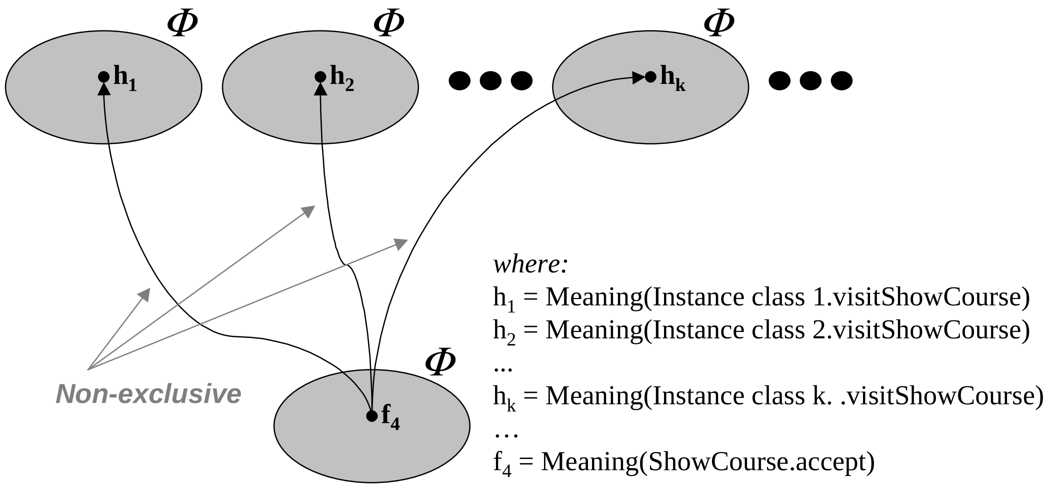
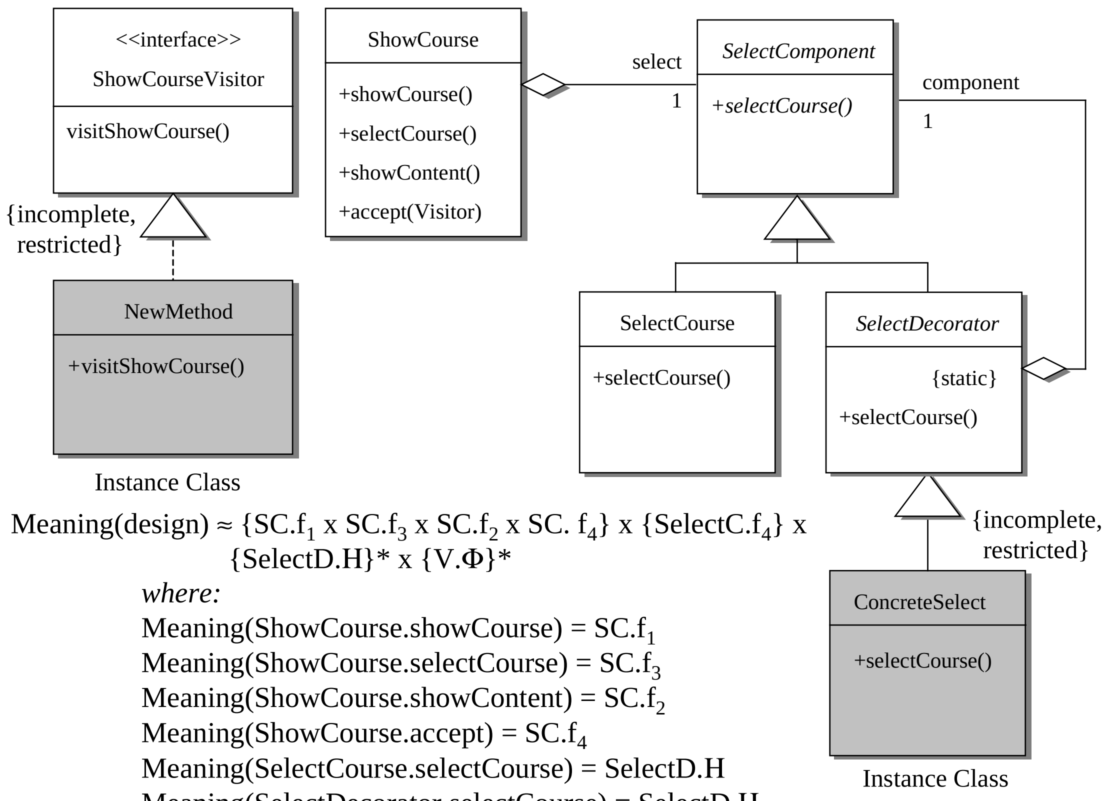

Φ
Φ
Φ
h1
h2
hk
Φ
f4
Non-exclusive
where:
h1 = Meaning(Instance class 1.visitShowCourse)
h2 = Meaning(Instance class 2.visitShowCourse)
...
hk = Meaning(Instance class k. .visitShowCourse)
…
f4 = Meaning(ShowCourse.accept)

Figure 9.11. Geometric view of the extension class-visitor transformation

If the decorator pattern [34] (Figure 9.12) had been used in the place of the strategy, the transformation (i) expression would be:

$$\{SC.H\} = \{SC.f_3\} \times \{SelectC.f_4\} \times \{SelectD.H\}^*$$

Since *SelectComponent* is an abstract class it has no semantics. This approach is more appropriate to implement variation methods if there is a vanilla version of the method to which new features may be added. Decorator allows different subclasses to add different features to the default implementation. In this example, the *selectCourse()* method in class *SelectClass* provides a basic implementation to the course selection algorithm. To add new features to it instance classes of the *SelectDecorator* need to be defined. The overall scenario is the following: the *selectCourse()* method in class *ShowCourse* invokes a *selectCourse()* in one of the new instance classes, which in turn invokes the basic course selection procedure and also performs the new behavior.



Meaning(design) ≈ $\{SC.f_1 \times SC.f_3 \times SC.f_2 \times SC.\ f_4\} \times \{SelectC.f_4\} \times \{SelectD.H\}^* \times \{V.\Phi\}^*$

where:

Meaning(ShowCourse.showCourse) = $SC.f_1$
Meaning(ShowCourse.selectCourse) = $SC.f_3$
Meaning(ShowCourse.showContent) = $SC.f_2$
Meaning(ShowCourse.accept) = $SC.f_4$
Meaning(SelectCourse.selectCourse) = SelectD.H
Meaning(SelectDecorator.selectCourse) = SelectD.H
Meaning(ShowCourseVisitor.visitShowCourse) = $V.\ \Phi$
$H = \{h \mid \Pi\} \wedge H \subseteq \Phi$

Figure 9.12. Semantics of the decorator implementation for variation methods

Thus, when $SC.f_3$ is invoked, it invokes one of the available *SelectD.h*, which executes $SelectC.f_4$ and complements its behavior with the new features. Thus, the expression $\{SC.f_3\} \times \{SelectC.f_4\} \times \{SelectD.H\}^*$ may be reduced to $\{SC.f_4 o H\}$, which is the composition of the basic function f_4 and the added feature. If this composition belongs to H the transformation is proved to be behavior-preserving.

Note that decorator also allows for the addition of more than one feature, thus allowing for the invocation a chain of decorators before actually calling the basic function.

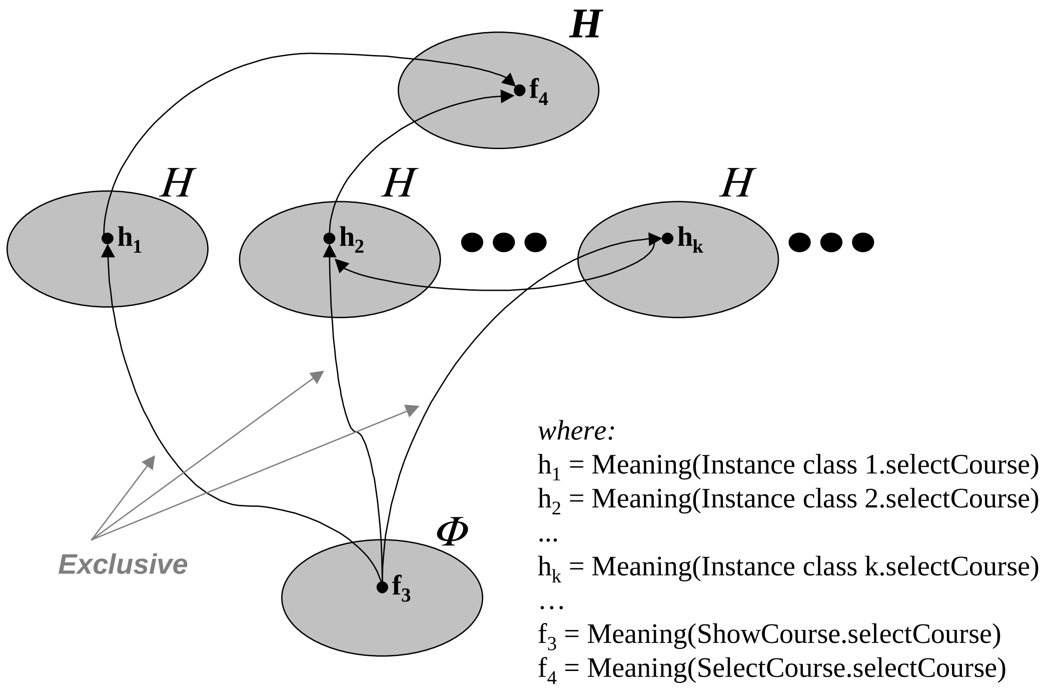

H
f4
H
H
H
h1
h2
hk
Φ
Exclusive
f3
where:
h1 = Meaning(Instance class 1.selectCourse)
h2 = Meaning(Instance class 2.selectCourse)
...
hk = Meaning(Instance class k.selectCourse)
…
f3 = Meaning(ShowCourse.selectCourse)
f4 = Meaning(SelectCourse.selectCourse)

Figure 9.13. Geometric view of the variation method-decorator transformation

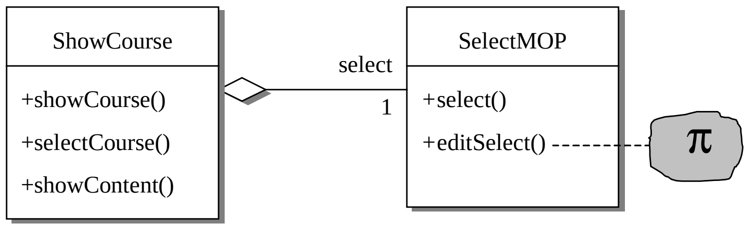
9.3.2 META-LEVEL PROGRAMMING TRANSFORMATIONS

As discussed in chapter 6, meta-object protocols [51] may be used to implement variation methods that have instantiation restrictions. Figure 9.14 shows how the variation method, *selectCourse()*, may be implemented by an MOP and presents the symbolic representation of the design semantics.

The variation method-MOP transformation is a behavior-preserving transformation if:

$$\{SC.H\} = \{MOP.f_5 \ x \ MOP.f_6\}$$

The *editSelect()* method is used to configure the course selection procedure, which is represented internally in the *SelectMOP* class. The selection procedures that may be configured by *editSelect()* are all the ones that satisfy the restrictions Π, which in this case is the entire set H. Every time *select()* is invoked it executes the current version of the course selection procedure. Since the set of procedures available is *H*, the *{MOP.*f_5 *x MOP.*f_6*}* expression may be reduced to *{MOP.H}*, which is equivalent to *{SC.H}*, concluding the proof. Figure 9.15 presents a geometric view of this mechanism.

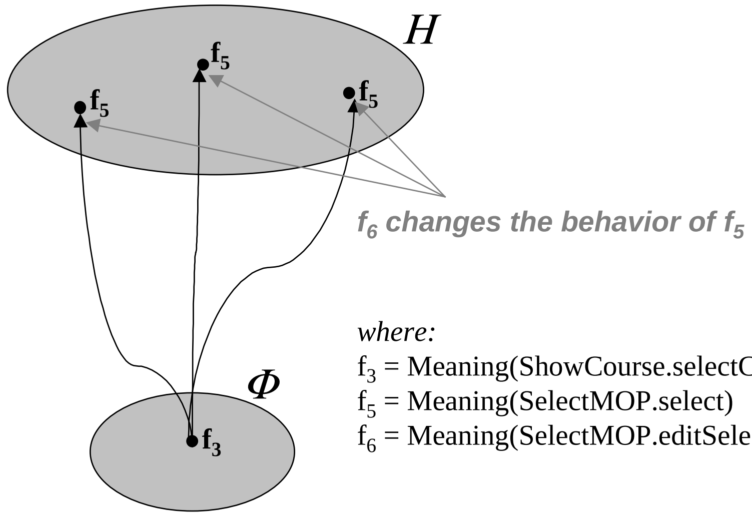


Meaning(design) ≈ {SC.f_1 x SC.f_3 x SC.f_2} x {MOP.f_5 x MOP.f_6}

where:

Meaning(ShowCourse.showCourse) = SC. f_1
Meaning(ShowCourse.selectCourse) = SC. f_3
Meaning(ShowCourse.showContent) = SC. f_2
Meaning(SelectMOP.select) = MOP.f_5
Meaning(SelectMOP.editSelect) = MOP.f_6

Figure 9.14. Semantics of meta-object protocols



where:
f_3 = Meaning(ShowCourse.selectCourse)
f_5 = Meaning(SelectMOP.select)
f_6 = Meaning(SelectMOP.editSelect)

Figure 9.15. Geometric view of the MOP mechanism to implement variation methods

9.3.3 AOP TRANSFORMATIONS

Figure 9.16 illustrates the implementation approach based on aspect-oriented programming [50, 57] that may be used to implement the web-based education framework, and presents the symbolic representation of its design semantics. To prove the soundness of this transformation the following expressions must be verified:

$$\{SC.f_1 \ x \ SC.H \ x \ SC.f_2 \ x \ SC.\Phi^*\} = \{SC.f_1 \ x \ SC.f_2\} \ x \ \{SelectA.H\}^* \ x \ \{ShowA.\Phi\}^*$$

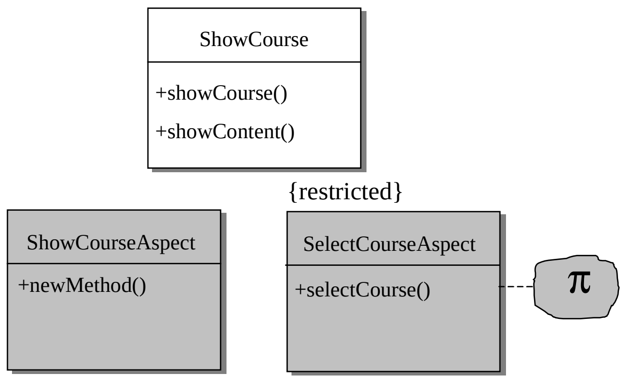
After the elimination of unused symbols the expression is simplified to:

$$\{SC.H x \ SC.\Phi^*\} = \{SelectA.H\}^* \ x \ \{ShowA.\Phi\}^*$$

Using the knowledge about which transformations had been applied, this expression can be divided into two expressions:

$$(i)\{SC.H\} = \{SelectA.H\}^* \text{ and } (ii) \{SC.\Phi^*\} = \{ShowA.\Phi\}^*$$

Transformation (i) represents the use of *after* aspects to implement variation methods. Since *after* aspect methods execute after the execution of a given method, only one *after* aspect may be invoked each time. This is assured by the application developer by plugging only one *after* aspect per hot-spot during the framework instantiation. Thus, the repetition symbol may be eliminated and the formula is proved to be sound. Transformation (ii) represents the use of *new* aspects to implement extension classes and can be automatically verified. Note that several new aspects may be used at the same time for a given class.



Meaning(design) ≈ $\{SC.f_1 \ x \ SC.f_2\} \ x \ \{SelectA.H\}^* \ x \ \{ShowA.\Phi\}^*$

where:

Meaning(ShowCourse.showCourse) = SC. f_1

Meaning(ShowCourse.showContent) = SC. F_2

Meaning(SelectCourseAspect.selectCourse) = SelectA.H

Meaning(ShowCourseAspect.newMethod) = ShowA.Φ

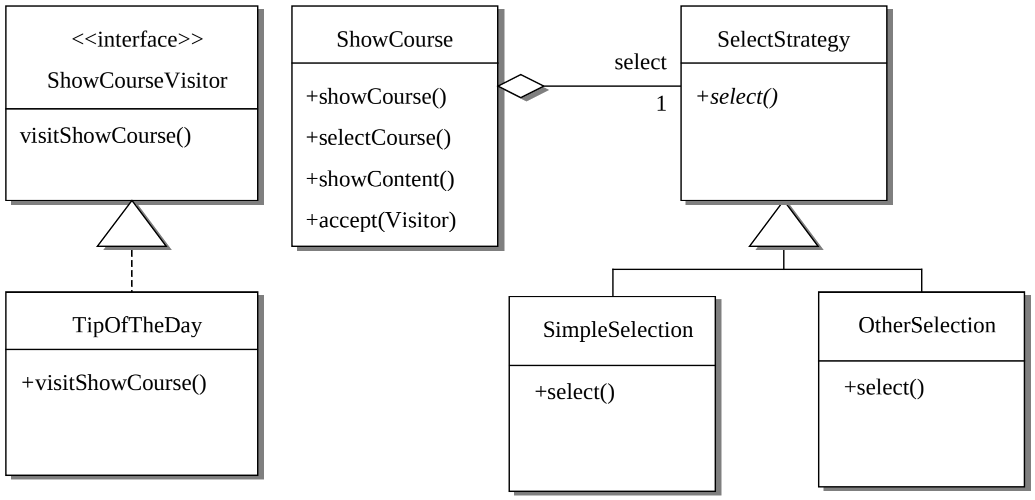
$H = \{h \mid \Pi\} \wedge H \subseteq \Phi$

Figure 9.16. AOP based implementation

The semantic verifications for the subject-oriented programming [38] approach are exactly the same the ones just presented for AOP.

9.4 INSTANTIATION TRANSFORMATIONS

Instantiation transformations transform the framework into concrete applications. They are not behavior-preserving, since *Meaning(application)* $\subset$ *Meaning(framework).* However, these transformations may be checked through the *InInstance* function. Figure 9.17 illustrates one possible instance of the Figure 9.9 framework and the symbolic representation of their semantics.



Meaning(design) ≈ $\{SC.f_1 \text{ x } SC.f_3 \text{ x } SC.f_2 \text{ x } SC.f_4\} \text{ x } \{SS.f_5\} \text{ x } \{OS.f_6\} \text{ x } \{TD.f_7\}$

where:

Meaning(ShowCourse.showCourse) = $SC.f_1$
Meaning(ShowCourse.selectCourse) = $SC.f_3$
Meaning(ShowCourse.showContent) = $SC.f_2$
Meaning(ShowCourse.accept) = $SC.f_4$
Meaning(SimpleSelection.select) = $SS.f_5$
Meaning(OtherSelection.select) = $OS.f_6$
Meaning(TipOfTheDay.visitShowCourse) = $TD.f_7$

Figure 9.17. Semantics of a web-based education application

To prove that it is a valid instance, the following equation must be proved:

$$\{SC.f_1 \text{ x } SC.f_3 \text{ x } SC.f_2 \text{ x } SC.f_4\} \text{ x } \{SS.f_5\} \text{ x } \{OS.f_6\} \text{ x } \{TD.f_7\} \subset \{SC.f_1 \text{ x } SC.f_3 \text{ x } SC.f_2 \text{ x } SC.f_4\} \text{ x } \{SS.H\}^* \text{ x } \{V.\Phi\}^*$$

It can be automatically simplified to:

$$\{SS.f_5\} \times \{OS.f_6\} \times \{TD.f_7\} \subset \{SS.H\}^* \times \{V.\Phi\}^*$$

Knowing what instance classes have been added to what hot-spots, the equation may be simplified to two simpler ones:

$$(i)\{SS.f_5\} \times \{OS.f_6\} \subset \{SS.H\}^* \text{ and } (ii)\ \{TD.f_7\} \subset \{V.\Phi\}^*$$

Equation (ii) holds since every method implementation belongs to Φ. Equation (i), on the other hand, only holds if $\{f_5\ ,\ f_6\} \subset H$, which cannot be verified automatically since checking for properties in programs is generally an undecidable task. However, a tool can always list in polynomial time the set of equations that must be true for the design to be considered a framework instance. The *verifier tool* works that way, as will be shown in section 9.6.

However, if the instantiation diagram presented in Figure 9.18 is the one that defines the instantiation process of the framework presented in Figure 9.9, the application shown in Figure 9.17 would not be a valid framework instance. The reason is that this instantiation diagram limits *SelectStrategy* to only one subclass.

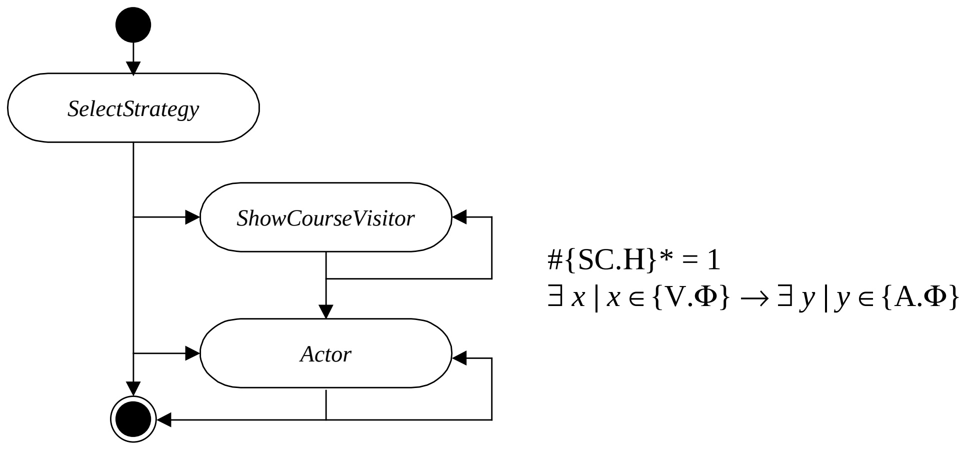


Figure 9.18. Instantiation diagram of the web-based education framework

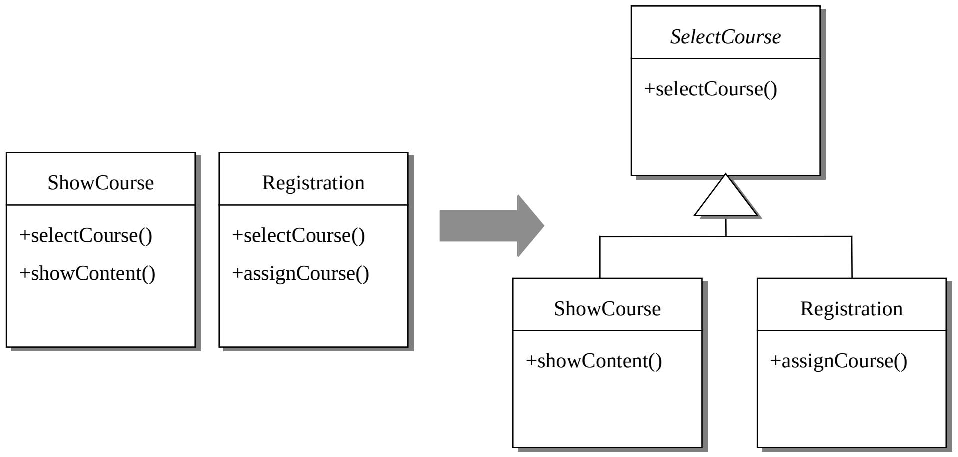
Instantiation diagrams may be represented as first-order logic specifications on the sets that define the framework semantics, as shown in Figure 9.18. In this example, the following specification should be verified in order for an application to be considered a framework instance: *SelectStrategy* must have only one subclass and if *ShowCourseVisitor* is instantiated Actor must also be. Although this specification does not make sense, since *Actor* and *ShowCourseVisitor* are not related hot-spots, it is what the instantiation diagram implies.

The approaches to framework instantiation described in chapter 7 assure that the instantiation diagrams are always respected.

9.5 MAINTENANCE TRANSFORMATIONS

Refactorings [47, 66] are behavior-preserving transformations, as shown in chapter 8. This section shows that some of the refactoring transformations presented throughout the dissertation really preserve the design semantics.

Figure 9.19 illustrates a refactoring that moves common behavior to abstract super classes, showing the semantics of the design before and after the transformation.



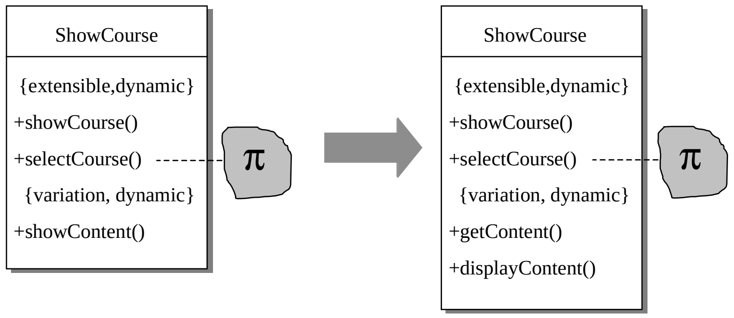
Meaning(before) ≈ $\{SC.f_1 \text{ x } SC.f_2\} \text{ x } \{R.f_1 \text{ x } R.f_3\}$
Meaning(after) ≈ $\{SC.f_1 \text{ x } SC.f_2\} \text{ x } \{R.f_1 \text{ x } R.f_3\}$

where:
Meaning(ShowCourse.selectCourse) = $SC.f_1$
Meaning(ShowCourse.showContent) = $SC.f_2$
Meaning(ShowCourse.selectCourse) = $SC.f_1$
Meaning(ShowCourse.assignCourse) = $SC.f_3$

Figure 9.19. Moving common methods to abstract super classes

This transformation is sound and it can be automatically verified by a tool whether it is true that the methods being moved have the same behavior. Note that this refactoring pre-condition is undecidable and cannot be automatically verified.

The semantics split method transformation is presented in Figure 9.20. A tool can say in polynomial time that this refactoring is sound if *getContent()* and *displayContent()* together have the same semantics as *showContent()* (*$\{SC.f_2\} = \{SC.f_3 \text{ x } SC.f_4\}$*). However, this verification cannot be made automatically. Several other refactorings can be easily verified, such as renaming and converting the access status of methods (from public to private, for example).



Meaning(before) ≈ $\{SC.f_1 \times SC.H \times SC.f_2 \times SC.\Phi^*\}$
Meaning(after) ≈ $\{SC.f_1 \times SC.H \times SC.f_3 \times SC.f_4 \times SC.\Phi^*\}$
where:

Meaning(ShowCourse.showCourse) = $SC.\ f_1$
Meaning(ShowCourse.selectCourse) = $SC.\ H$
Meaning(ShowCourse.showContent) = $SC.\ F_2$
Meaning(ShowCourse.getContent) = $SC.\ f_3$
Meaning(ShowCourse.displayContent) = $SC.\ f_4$
$H = \{h \mid \Pi\} \wedge H \subseteq \Phi$

Figure 9.20. Splitting method transformation and its semantics

Unlike refactoring rules, unification rules [2] are not behavior-preserving. They are used to avoid the architectural drift problem [16] by changing the hot-spot structure of a system. If the design modeled by *application* is not a valid instance of a *framework*, the unification rules may be applied to change that. They modify the *framework* design to include *application* in the set of possible instances. Figure 9.21 shows geometrically the meaning of a unification transformation.

To formally verify a unification procedure it must be checked that before the transformation *IsInstance(framework, application)* does not hold and that it holds after the transformation.

Figure 9.22 illustrates a simple framework and Figure 9.23 shows an application that cannot be created from it. Even if $\{f_5\ ,\ f_6\} \subset H$, *IsInstance* fails since $SC.f_4 \not\subset$ *Meaning(framework).* In this situation the unification procedure that creates extension classes can be applied to transform the framework design into the one shown in Figure 9.24. Note that if the extension class-visitor transformation is applied, the framework design shown in Figure 9.9 is generated.

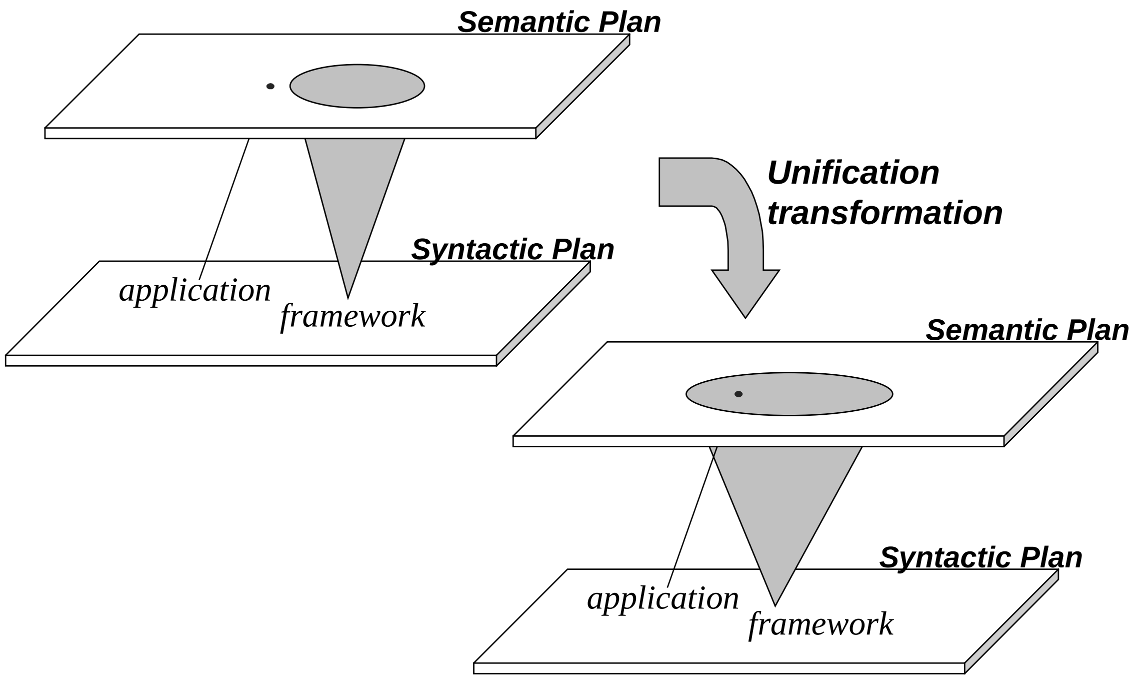
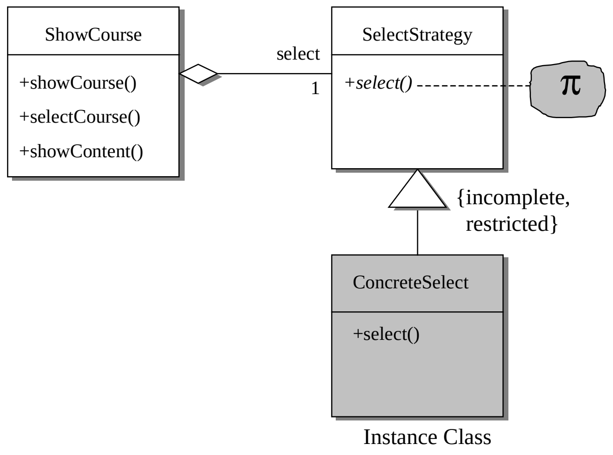


Figure 9.21. Geometric view of unification transformations

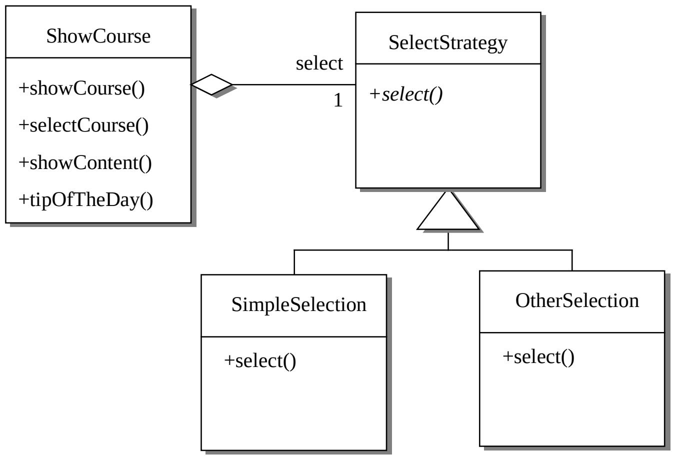
After the unification transformation the *IsInstance* function is verified if $\{f_5 , f_6\} \subset H$ since $f_4 \in \Phi$.



Meaning(framework) ≈ $\{SC.f_1 \times SC.f_2 \times SC.f_3\} \times \{SS.\ H\}^*$
where:

Meaning(ShowCourse.showCourse) = $SC.\ f_1$
Meaning(ShowCourse.selectCourse) = $SC.\ f_2$
Meaning(ShowCourse.showContent) = $SC.\ f_3$
Meaning(SelectStrategy.select) = $SS.\ H$
$H = \{h \mid \Pi\} \wedge H \subseteq \Phi$

Figure 9.22. Framework design before the unification transformation

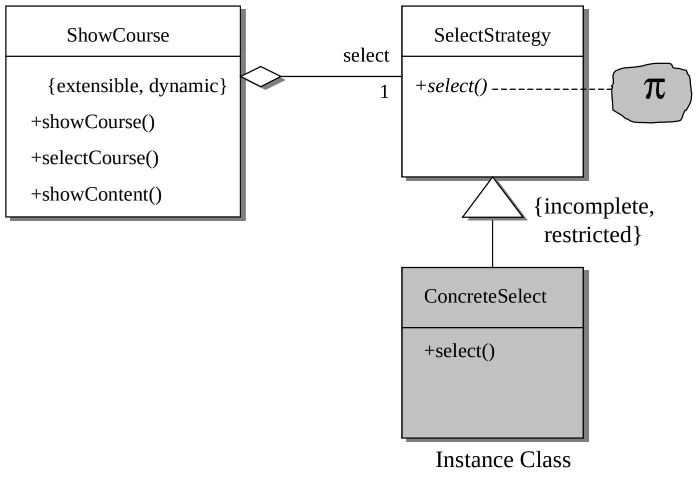


Meaning(application) ≈ {$SC.f_1$ x $SC.f_2$ x $SC.f_3$ x $SC.f_4$} x {$SS.f_5$} x {$OS.f_6$}

where:

Meaning(ShowCourse.showCourse) = $SC.f_1$
Meaning(ShowCourse.selectCourse) = $SC.f_3$
Meaning(ShowCourse.showContent) = $SC.f_2$
Meaning(ShowCourse.tipOfTheDay) = $SC.f_4$
Meaning(SimpleSelection.select) = $SS.f_5$
Meaning(OtherSelection.select) = $OS.f_6$

Figure 9.23. Invalid application



Meaning(framework) ≈ {$SC.f_1$ x $SC.f_2$ x $SC.f_3$ x $SC.\Phi^*$} x {SS. H}*

where:

Meaning(ShowCourse.showCourse) = SC. f_1
Meaning(ShowCourse.selectCourse) = SC. f_2
Meaning(ShowCourse.showContent) = SC. f_3
Meaning(SelectStrategy.select) = SS. H
$H = \{h \mid \Pi\} \wedge H \subseteq \Phi$

Figure 9.24. Framework design after the unification transformation

9.6 THE VERIFIER TOOL

The *verifier tool* can be used anytime during the framework development life-cycle to check transformations. Since new implementation models, refactorings, and unifications may be configured, the use of *verifier* is important to ensure that the transformations used are sound.

There are three predicates that may be used by the *verifier* users:

- *meaning(Project):* uses the algorithm described in section 9.2 to calculate the semantics of Project. Figure 9.25 illustrates the use of meaning to the framework presented in Figure 9.8;

```
?- meaning(aladin).
{[[showCourse.showCourse]]
x[[showCourse.H1]]
x[[showCourse.showContent]]
x[[showCourse.Fi*]]}

Yes
```

Figure 9.25. Meaning example

- *isInstance(Project1, Project2):* checks if Project2 is an instance of Project1. In the case that some of the Project1 hot-spots have instantiation restrictions, it shows the conditions that must hold for the instance to be valid. It also verifies the instantiation diagrams, since they might limit the set of valid instances as shown in section 9.4. Figure 9.26 illustrates an application of this predicate to the design pattern based transformations described in Figure 9.9;

```
?- isInstance(dp, instance1).
If:
[[simpleSelection.select]] belongs to H1
and [[otherSelection.select]] belongs to H1

Yes
```

Figure 9.26. isInstance example

- *validImplementation(Project1, Project2):* uses the transformation history to assist its users in the proof of implementation diagrams. It compares the semantics of Project1 and Project2 and lists the conditions that must hold for the implementation transformation to be behavior-preserving. Its applicability to the example discussed in section 9.4 is illustrated in Figure 9.27;

```
?- validImplementation (aladin, dp).
If:
{[[showCourse.H1]]} = {[[showCourse.selectCourse]]} x
{[[selectStrategy.H1]]}*
and {[[showCourse.Fi*]]} = {[[showCourse.accept]]} x
{[[ShowCourseVisitorectStrategy.Fi*]]}*
 belongs to H1
[[otherSelection.select]] belongs to H1

Yes
```

Figure 9.27. validImplementation example

- *validRefactoring(Project1, Project2):* works as in *validImplementation* but verifies only refactoring transformations. Figure 9.28 shows its applicability to the splitting refactoring illustrated in section 9.5.

```
?- validImplementation(aladin1, aladin2).
If:
{[[showCourse.showContent]]} ={[[showCourse.getContent]]x
[[showCourse.displayContent]]}*

Yes
```

Figure 9.28. validRefactoring example

The predicates *validImplementation* and *validRefactoring* can be combined in one *sameMeaning* predicate. The distinction is to make the use of the transformation history explicit in the verification process.

9.7 SUMMARY

The approach for calculating the semantics of a design is based on set theory. The semantics of a design depends on the semantics of its classes, which in turn depend on the semantics of its methods. If a design has no hot-spots, the cardinality of the set that defines its semantics is one. On the other hand, if it has at least one hot-spot it can be defined as a framework. This definition provides an intuitive and precise meaning to what frameworks really are, which is still an open issue [46].

The semantics of a design can be calculated in polynomial time. Procedures that help the verification of the transformations presented throughout this dissertation may also be implemented in polynomial time. The *verifier* tool is an example of a tool that may be constructed to calculate the semantic of projects and verify transformations. It uses the transformation history to better assist the user in the verification process and to speed up the verification, since only the modified parts need to be checked.

Chapter 10 – Case Studies

This chapter presents five case studies on the use of the design language and its supporting tools for the analysis and development of real-world frameworks. The first two case studies are analyses of frameworks that have been developed without the support of the ideas presented in this dissertation. The two frameworks presented in this category are: Neighbor, for the local search heuristics domain and Unidraw, for the graphical object editors domain. The use of the design language and tools to support framework development is also illustrated by two case studies: PROC, for the process modeling domain and V-Market, for the virtual marketplaces domain. Finally, the unification-based development approach is illustrated by the description of the ALADIN framework development project.

10.1 NEIGHBOR

Hard combinatorial optimization problems usually have to be solved by approximate methods. Constructive methods build up feasible solutions from scratch. Among them we find the so-called greedy algorithms, based on a preliminary ordering of the solution elements according to their cost values. Basic local search methods are based on the evaluation of the neighborhood of successive improving solutions, until no further improvements are possible. As an attempt to escape from local optima, some methods allow controlled deterioration of the solutions in order to diversify the search [5].

In the study of heuristics for combinatorial problems, it is often important to develop and compare, in a systematic way, different algorithms, strategies, and parameters for the same problem. The *Neighbor* framework [5] encapsulates different aspects involved in local search heuristics, such as algorithms for the construction of initial solutions, methods for neighborhood generation, and movement selection criteria. Encapsulation and abstraction promote unbiased comparisons between different heuristics, code reuse, and easy extensions.

10.1.1 NEIGHBOR PATTERN-BASED DESCRIPTION

This section describes the framework as it was first documented by its authors [5], using a variation of the pattern form proposed in [34] and OMT diagrams [81].

Intent:

To provide an architectural basis for the implementation and comparison of different local search strategies.

Motivation:

In the study of heuristics for combinatorial problems, it is important to develop and compare, in a systematic way, different heuristics for the same problem. It is frequently the case that the best strategy for a specific problem is not a good strategy for another. It follows that, for a given problem, it is often necessary to experiment with different heuristics, using different strategies and parameters.

By modeling the different concerns involved in local search in separate classes, and relating these classes in a framework, our ability to construct and compare heuristics is increased, independently of their implementations. Implementations can easily affect the performance of a new heuristic, for example due to programming language, compiler, and other platform aspects.

Applicability:

The *Neighbor* framework can be used in situations involving:

- Local search strategies that can use different methods for the construction of the initial solution, different neighborhood relations, or different movement selection criteria;
- Construction algorithms that utilize subordinate local search heuristics;
- Local search heuristics with dynamic neighborhood models.

Structure:

Figure 10.1 shows the classes and relationships involved in the *Neighbor* framework.

Participants:

- *Client:* contains the combinatorial problem instance to be solved, its initial data and the pre-processing methods to be applied. It also contains the data for creating the *SearchStrategy* that will be used to solve the problem. Generally, it can have methods for processing the solutions obtained by the *SearchStrategy*.
- *Solution:* encapsulates the representation of a solution for the combinatorial problem. It defines the interface the algorithms must use in order to construct and modify a

solution. It delegates to *IncrementModel* or to *MovementModel* requests to modify the current solution.

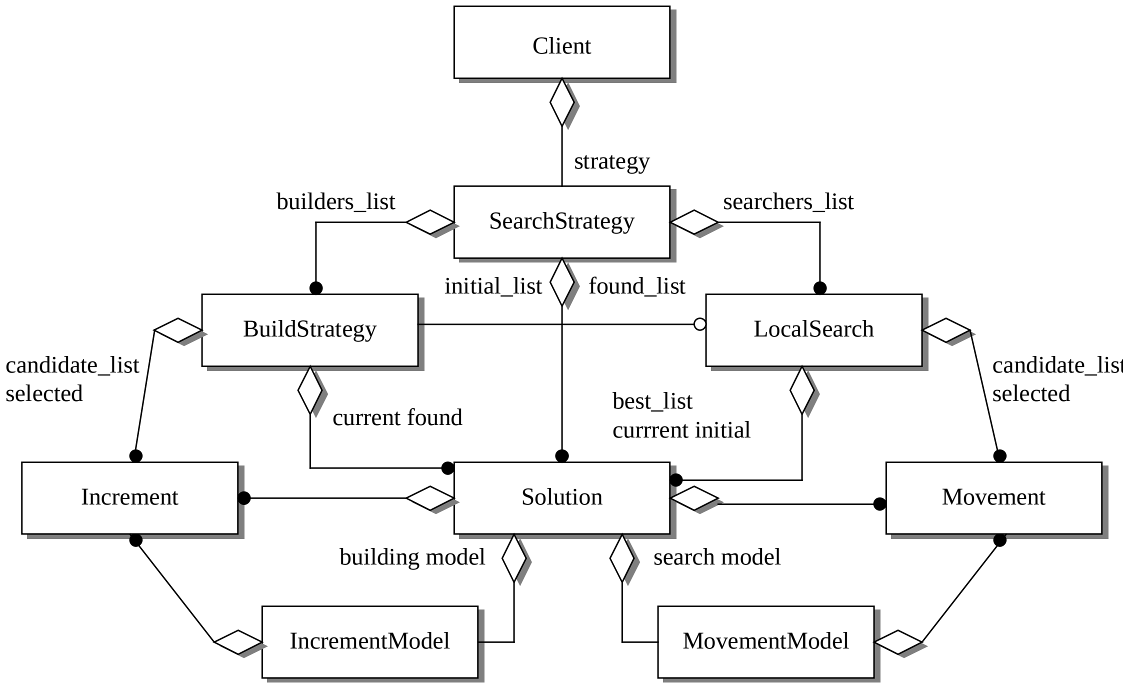


Figure 10.1. Neighbor class diagram

- *SearchStrategy:* constructs and starts the *BuildStrategy* and the *LocalSearch* algorithms, also handling their intercommunication, in case it exists.
- *BuildStrategy:* encapsulates constructive algorithms in concrete subclasses. It investigates and eventually requests *Solution* for modifications in the current solution, based on an *IncrementModel*.
- *LocalSearch:* encapsulates local search algorithms in concrete subclasses. It investigates and eventually requests *Solution* for modifications in the current solution, based on a *MovementModel*.
- *Increment:* groups the data necessary for an atomic modification of the internal representation of a solution for constructive algorithms.
- *Movement:* groups the data necessary for an atomic modification of the internal representation of a solution for local search algorithms.
- *IncrementModel:* modifies a solution according to a *BuildStrategy* request.
- *MovementModel:* modifies a solution according to a *LocalSearch* request.

Collaborations:

The *Client* wants a *Solution* for a problem instance. It delegates this task to its *SearchStrategy*, which is composed by at least one *BuildStrategy* and one *LocalSearch.* The *BuildStrategy* produces an initial *Solution* and the *LocalSearch* improves the initial *Solution* through successive movements. The *BuildStrategy* and the *LocalSearch* perform their tasks based on neighborhood relations provided by the *Client*.

The implementation of these neighborhoods is delegated by the *Solution* to its *IncrementModel* (related to the *BuildStrategy*) and to its *MovementModel* (related to the *LocalSearch*). The *IncrementModel* and the *MovementModel* are the objects that will obtain the *Increment*s or the *Movement*s necessary to modify the *Solution* (under construction or not).

The *IncrementModel* and the *MovementModel* may change at run-time, reflecting the use of a dynamic neighborhood in the *LocalSearch*, or having a *BuildStrategy* that uses several kinds of *Increment*s to construct a *Solution*. The variation of the *IncrementModel* is controlled inside the *BuildStrategy* and the variation of the *MovementModel* is controlled by the *LocalSearch*. This control is performed using information made available by the *Client* and accessible to these objects. Figure 10.2 illustrates this scenario.

10.1.2 NEIGHBOR DESCRIPTION REVISITED

Although the pattern-based description gives an intuition of the framework design structure there are several problems related to it:

- Hot-spot identification: there is no indication in the diagrams of the framework hot-spots and how they need to be instantiated. The textual description is informal and might not be clear enough;
- Complex design model: the design diagram presented is very tangled and hard to be understood;
- Interrelated hot-spots: there are hot-spot interdependencies that are not represented in the diagrams. Whenever new build strategies are defined new increment models also need to be. The same holds for the search strategies and the movement models.

Figure 10.3 represents the Neighbor design structure through an extended class diagram while Figure 10.4 illustrates its instantiation diagram. Note that now the hot-spots are explicitly documented, the design is more clear (four classes have been eliminated) and the hot-spot interdependencies are formally documented in the instantiation diagram.

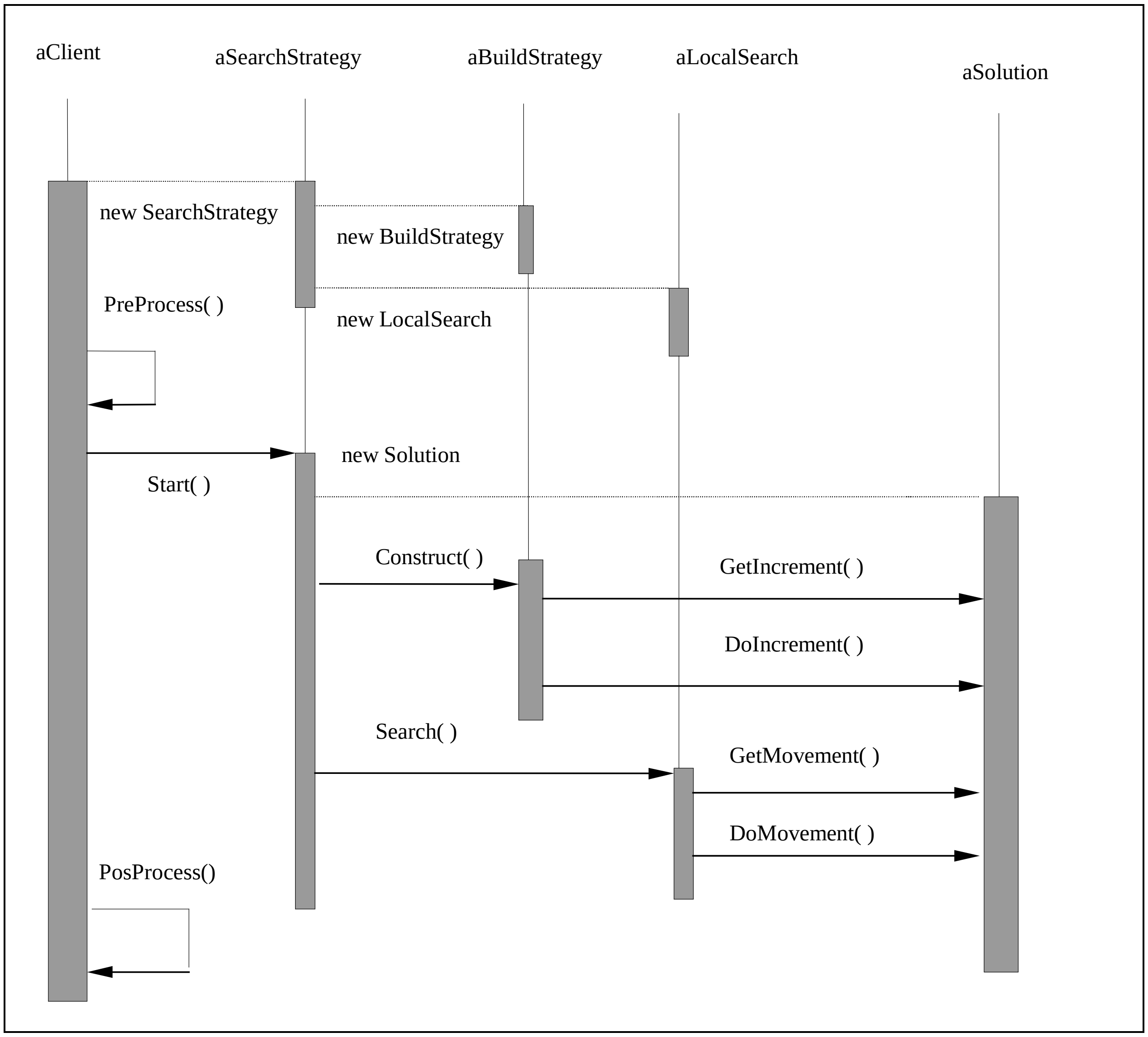
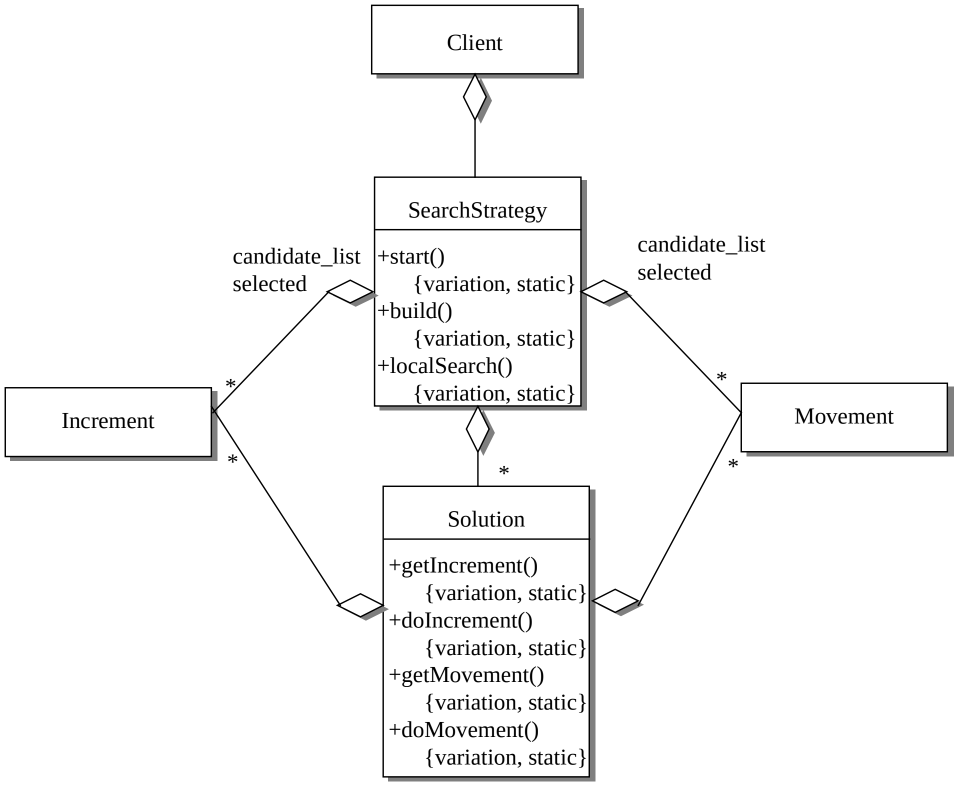


Figure 10.2. Collaborations in the Neighbor framework

If an implementation model that applies the strategy design pattern to all variation methods is used in the design diagram shown in Figure 10.3 an implementation diagram almost identical to the one presented in Figure 10.1 is generated, showing that the two models have the same semantics (since the strategy transformation preserves semantics as shown in chapter 9). The only difference is that the extension interfaces would be explicitly represented.

Figure 10.4 shows the instantiation trace design analysis applied to the Neighbor framework. Note that the instantiation traces that do appear are invalid. An example of an invalid instantiation is the definition of an increment model without the definition corresponding previous definition of a build strategy. This information cannot be provided to the framework user unless instantiation diagrams are defined.



Client
SearchStrategy
+start() {variation, static}
+build() {variation, static}
+localSearch() {variation, static}
candidate_list selected
candidate_list selected
Increment
Movement
Solution
+getIncrement() {variation, static}
+doIncrement() {variation, static}
+getMovement() {variation, static}
+doMovement() {variation, static}
*
*
*
*
*

Figure 10.3. UML extended class diagram for the Neighbor framework

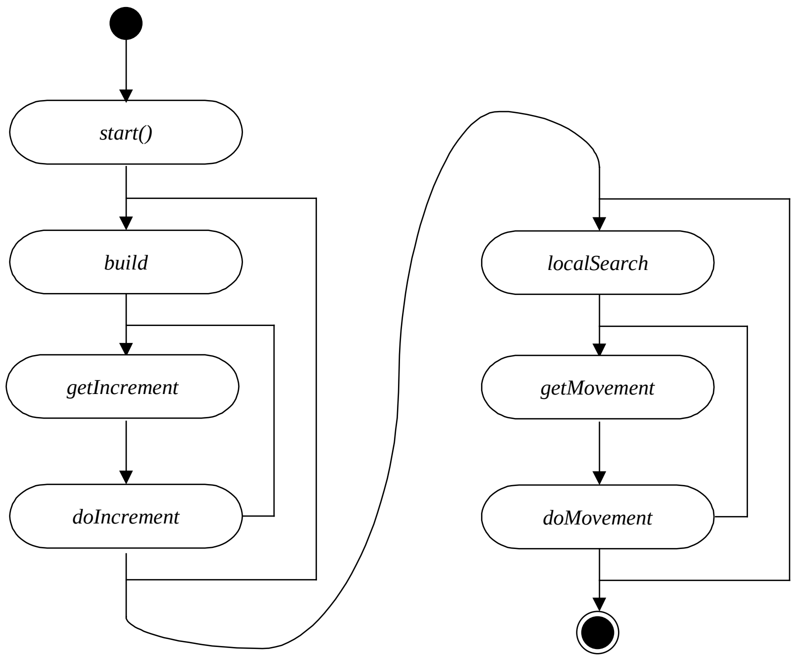


Figure 10.4. Neighbor instantiation diagram

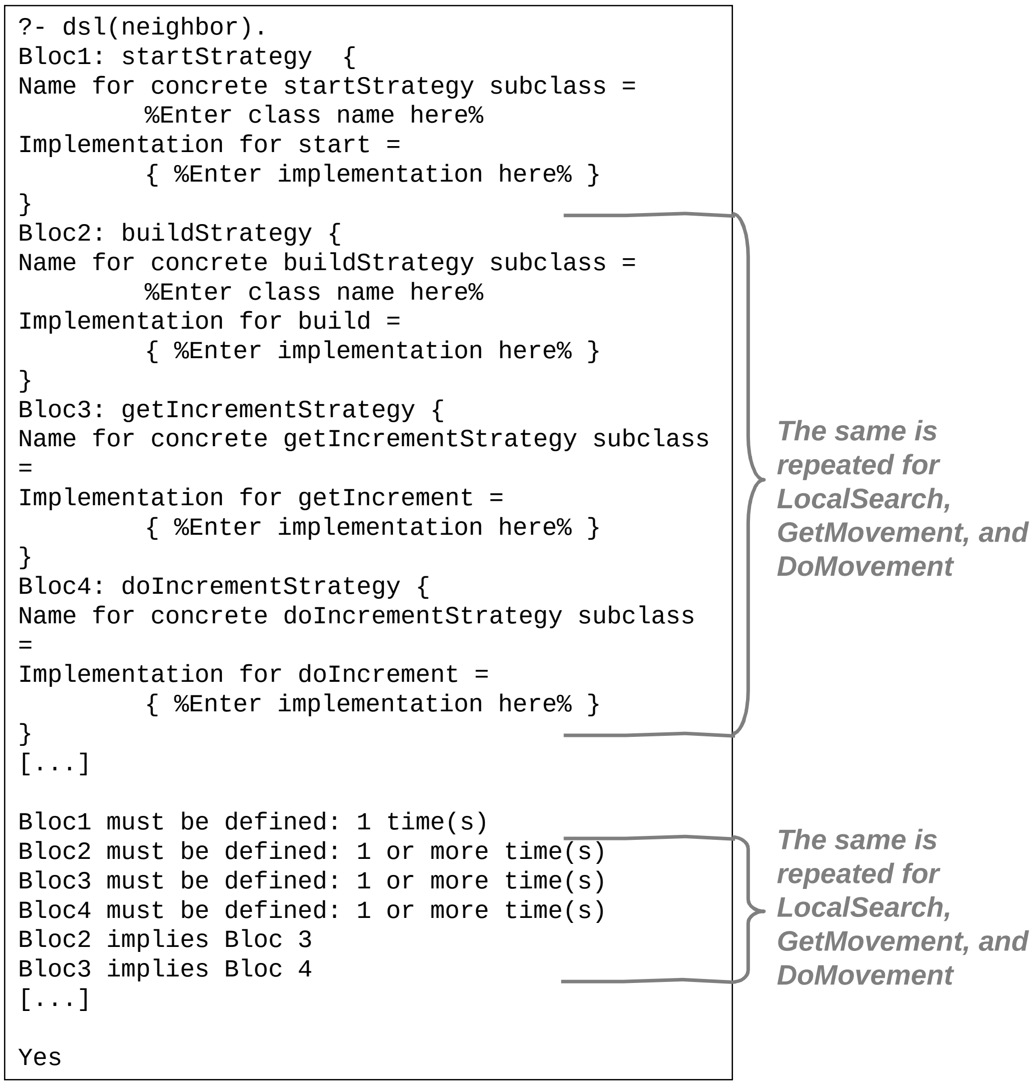
```
?- instantiationTrace(neighbor).

start, (build, (getIncrement, doIncrement)+)+,
(localSearch, (getMovement, doMovement)+)+

Yes                          At least one
```

Figure 10.5. Neighbor instantiation trace

An experiment that might be developed is the creation of a DSL compiler to the Neighbor framework. The DSL description defined by the *DSL generator* is presented in Figure 10.6. The DSL compiler might enhance a lot the instantiation process, which is currently based on the direct addition of code to the framework C++ program.



```
?- dsl(neighbor).
Bloc1: startStrategy  {
Name for concrete startStrategy subclass =
         %Enter class name here%
Implementation for start =
         { %Enter implementation here% }
}
Bloc2: buildStrategy {
Name for concrete buildStrategy subclass =
         %Enter class name here%
Implementation for build =
         { %Enter implementation here% }
}
Bloc3: getIncrementStrategy {
Name for concrete getIncrementStrategy subclass
=
Implementation for getIncrement =
         { %Enter implementation here% }
}
Bloc4: doIncrementStrategy {
Name for concrete doIncrementStrategy subclass
=
Implementation for doIncrement =
         { %Enter implementation here% }
}
[...]

Bloc1 must be defined: 1 time(s)
Bloc2 must be defined: 1 or more time(s)
Bloc3 must be defined: 1 or more time(s)
Bloc4 must be defined: 1 or more time(s)
Bloc2 implies Bloc 3
Bloc3 implies Bloc 4
[...]

Yes
```

The same is repeated for LocalSearch, GetMovement, and DoMovement
The same is repeated for LocalSearch, GetMovement, and DoMovement

Figure 10.6. Neighbor DSL

10.2 UNIDRAW

Unidraw [96] is a graphical editor framework that allows the construction of domain-specific editors. Different domain-specific editors normally require new graphical components. Unidraw allows for the definition of these new components through the creation of composite components *(GraphicComp* subclasses) from the set of primitive components *(Graphic* subclasses) and composite components previously defined in the system.

Figure 10.7 illustrates this design structure using the UML extended framework design notation. During instantiation, new subclasses of *Graphic* and *GraphicComp* may be defined. *Graphic* subclasses implement the framework primitive components while *GraphicComp* subclasses define composite components. In this example, the composite components are AND, OR, and NAND gates used to model electrical circuits in schematic capture systems.

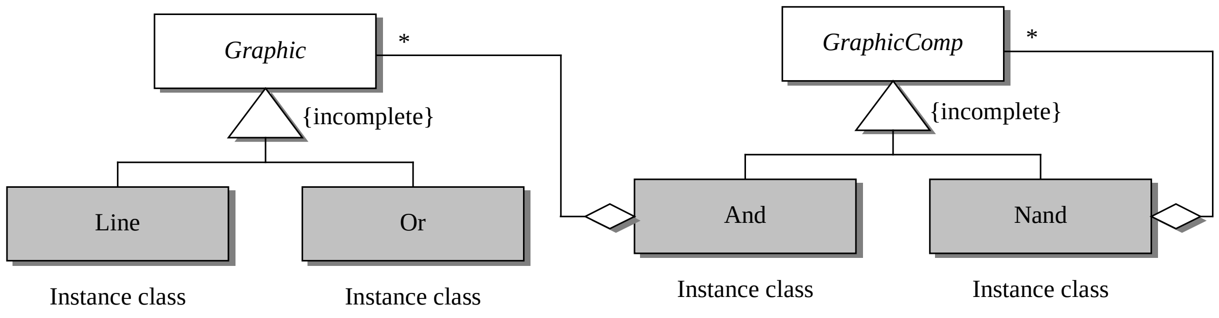


Figure 10.7. Unidraw Graphic-GraphicComp design structure

There is a balance between reuse and run-time performance that the Unidraw user should take into account when defining a new graphic component.

1. If he or she wants maximal reuse, a new component might be defined as a composition of existing *GraphicComp* subclass. The *Nand* component illustrated in Figure 10.7 uses this approach. This option may have performance problems due to the visualization approach adopted by the framework;
2. If he or she wants maximal performance, a new component should be described as a new primitive component *(Graphic* subclass). The *Or* component showed in Figure 10.7 uses this approach. This option will require much more implementation effort, since nothing is being reused;
3. An intermediate solution is the definition of a new component by a custom composition of existing *Graphic* subclasses, as in the case of the *And* component (Figure 10.7).

Note that options 1 and 3 are instantiated in the same way except for the implementation of the *draw()* method, which in the intermediate solutions should refer to *Graphic* subclasses. This is handled in the language by using template diagrams combined with instantiation diagrams, as shown in Figure 10.8. These diagrams formally guide the framework users in the instantiation of new Unidraw applications. The Unidraw original documentation [96] does not explicitly represents its hot-spots and how they should be instantiated, making it very hard for application developers to instantiate it properly.

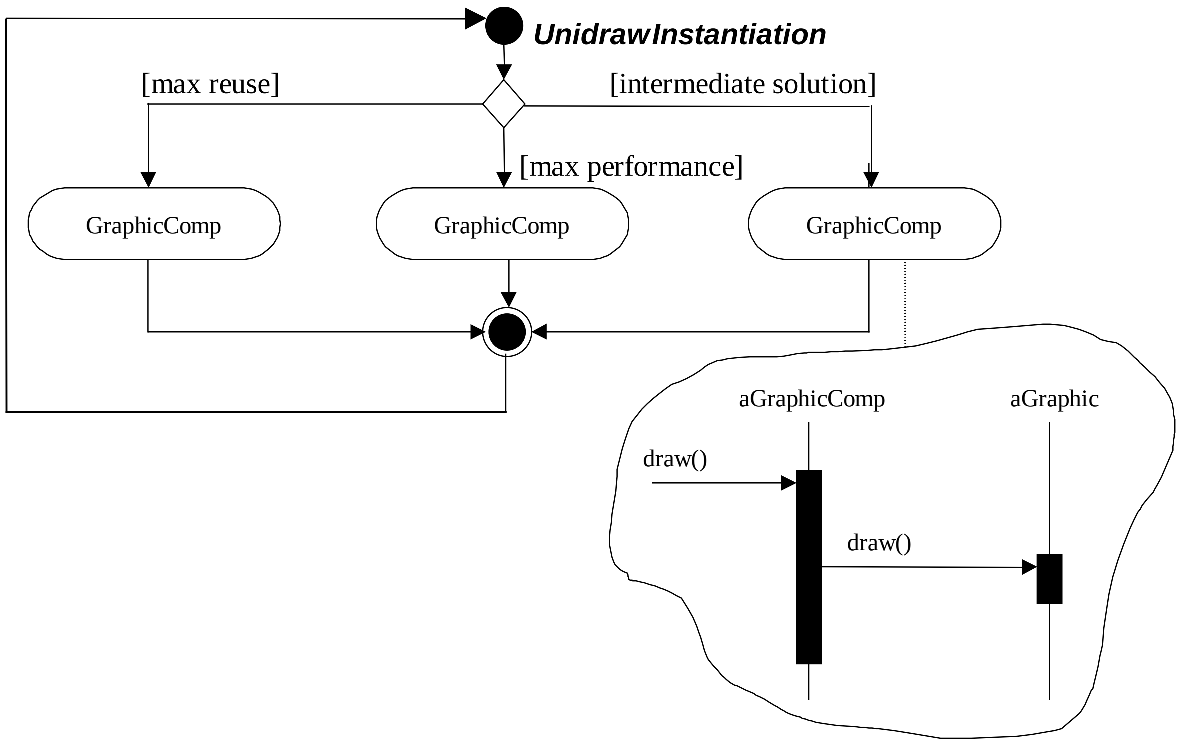
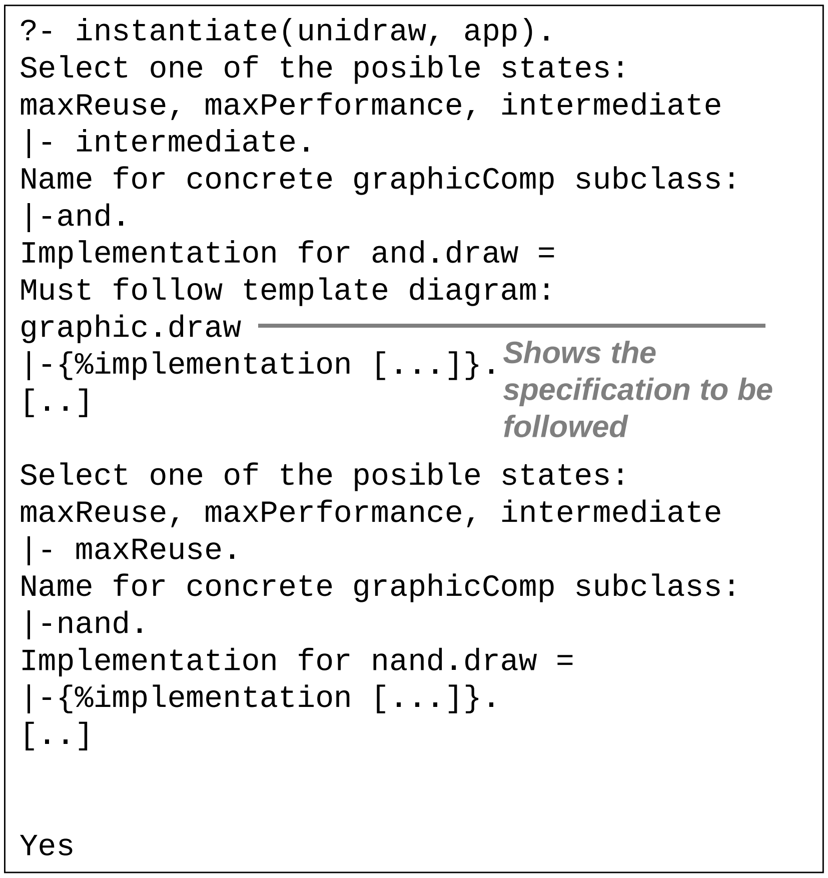


Figure 10.8. Unidraw Graphic-GraphicComp instantiation diagram

The necessity of adding instantiation restrictions to interaction diagram states has been noticed during the study of the Unidraw framework. There is no way to properly model this example if this capability is not added to the design language. The *process-based instantiation tool* functionality has also been extended to support this kind of specification. Figure 10.9 illustrates its use to guide the instantiation of this part of the Unidraw framework.



```
?- instantiate(unidraw, app).
Select one of the posible states:
maxReuse, maxPerformance, intermediate
|- intermediate.
Name for concrete graphicComp subclass:
|-and.
Implementation for and.draw =
Must follow template diagram:
graphic.draw
|-{%implementation [...]}.
[..]

Select one of the posible states:
maxReuse, maxPerformance, intermediate
|- maxReuse.
Name for concrete graphicComp subclass:
|-nand.
Implementation for nand.draw =
|-{%implementation [...]}.
[..]


Yes
```

Figure 10.9. Using the process-based instantiation tool to instantiate Unidraw

10.3 PROC

PROC is a framework for the edition of (software) process activities based on a formal process language [30]. The language is based on hierarchical Petri Nets, in which each state represents either an atomic activity or an entire net. An atomic activity can be of two kinds: human-based or tool-based.

Human-based activities represent the interaction of some actor *Role* with an *Artifact*. Roles are hierarchical (each role can contain sub-roles). Roles may also contain several *Groups* or *Actors*. When a process is executed each *Role* that participates in an *Activity* must be bound either to an *Actor* or to a *Group*. During the execution of the activity the *Fields* of an artifact can be updated by the actor(s) assigned to the activity. In tool-based activities the artifact’s fields are updated by a *Tool*. Figure 10.10 illustrates a process description, in which each state is represented by a name and the number of tokens it consumes.

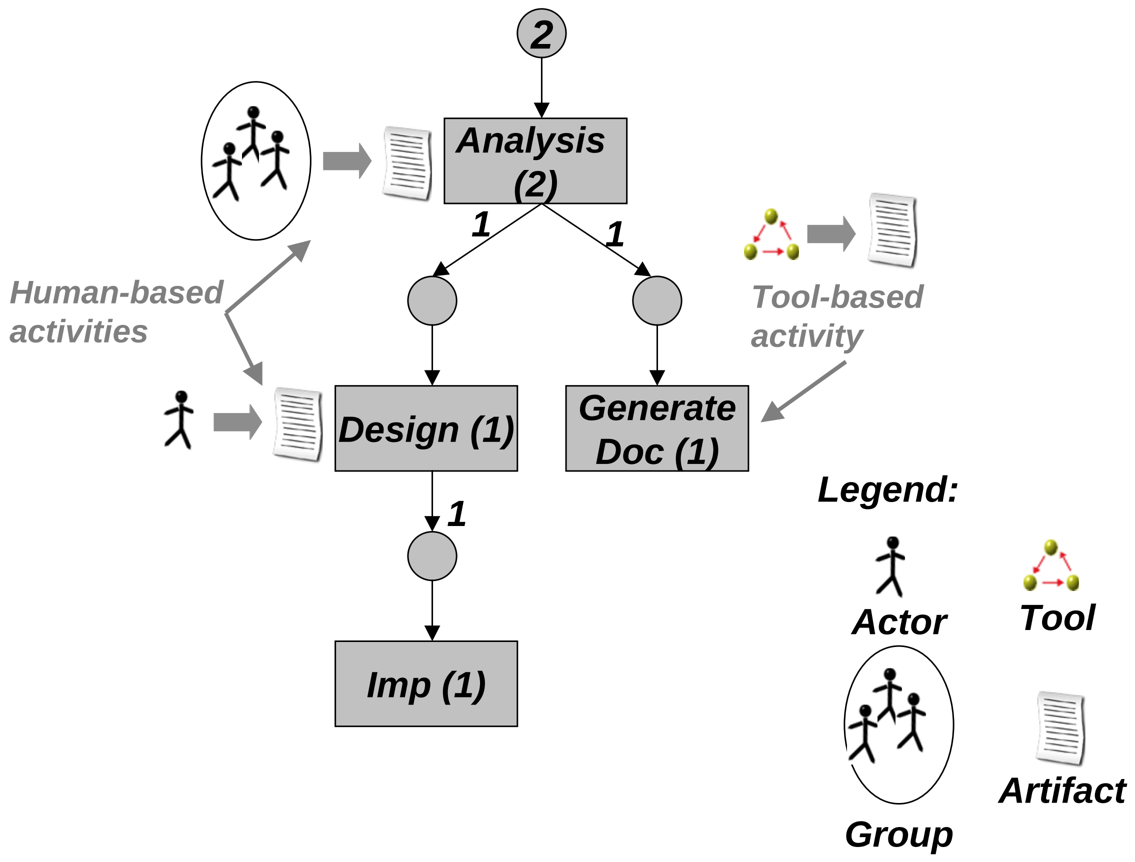


Figure 10.10. Process description example

The process execution is based on the Petri Net descriptions. Whenever a transition is enabled it is automatically fired. This can have two consequences: the actor(s)' *Agenda(s)* is (are) updated to include the new artifact that must be handled in the case of actor-based activities or a tool is executed in the case of tool-based activities. After the artifact fields are completed one of the actors assigned to the activity must inform the system that the activity is concluded. In the case of tool-based activities this is automatically performed by the tool. Figure 10.11 illustrates what happens during the execution of an human-based activity.

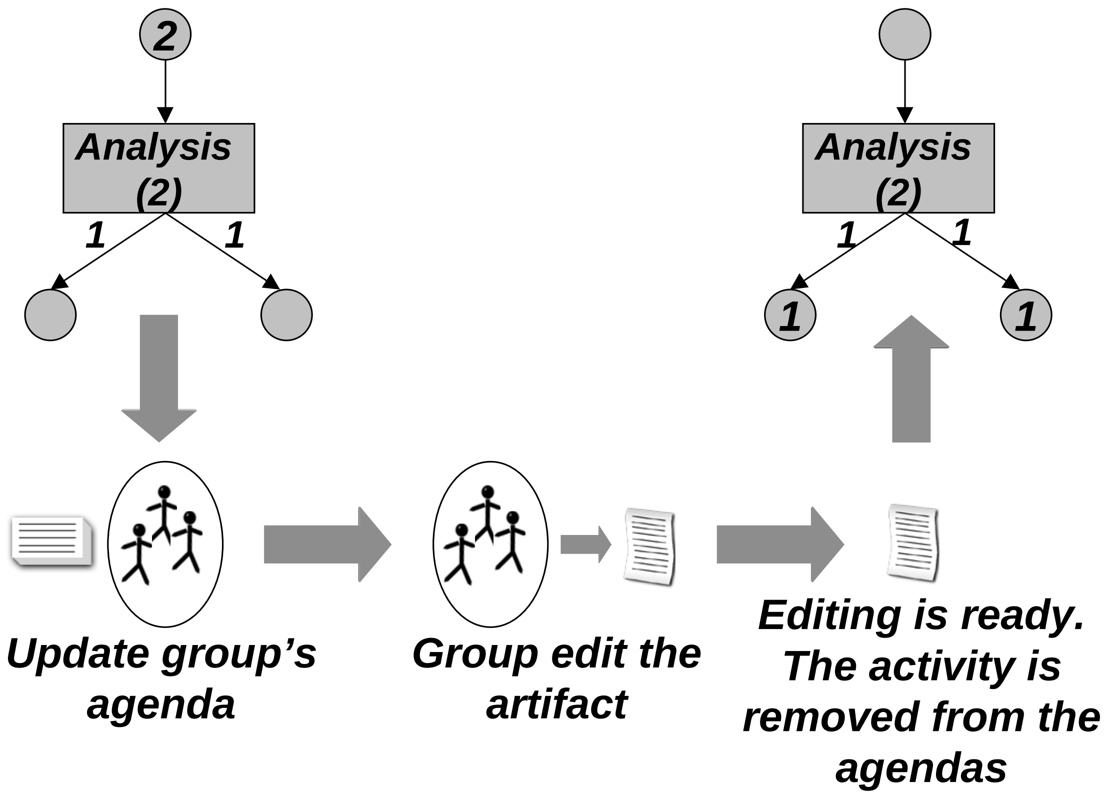


Figure 10.11. Executing the analysis activity

Pre and post-conditions can be assigned to each state as specifications on artifacts' fields. In the case that these execution constraints fail, an *Exception* is raised.

There are several hot-spots that must be completed in order to generate an instance of PROC.

Group, role, and actor properties: the properties that define an actor or a group may vary from one instantiation of PROC to another. Figure 10.12 illustrates these hot-spots;

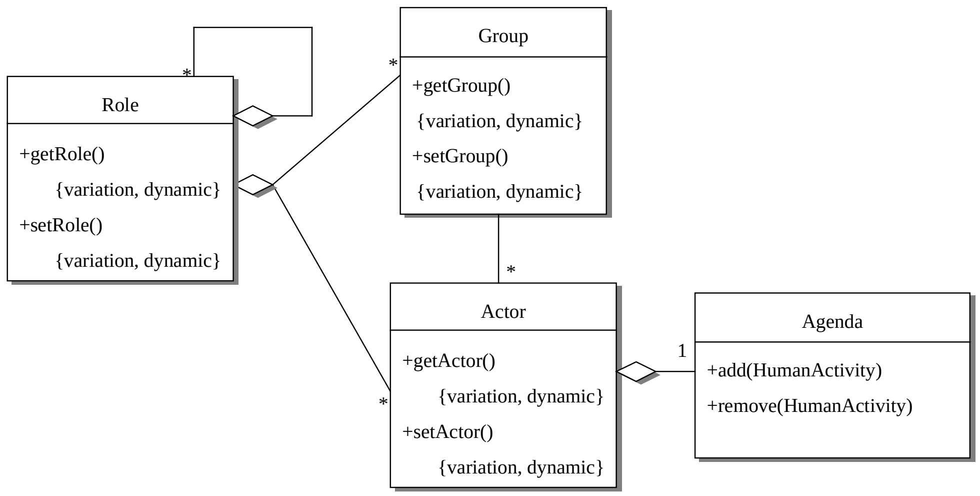


Figure 10.12. Group, role, and actor hot-spots

Artifact fields: artifacts are composed of artifact fields, and each field belongs to some field type. Since the pre and post-conditions are expressions on the artifact fields, whenever new field types are defined some comparison methods that may have to be used to verify the pre and post-condition specifications need to be defined. Figure 10.13 shows how activities and artifacts are defined in the system and how the field type hot-spot is modeled. Two variation methods are provided by the framework but new methods that operate on *FieldType* may be defined, making it an extension class;

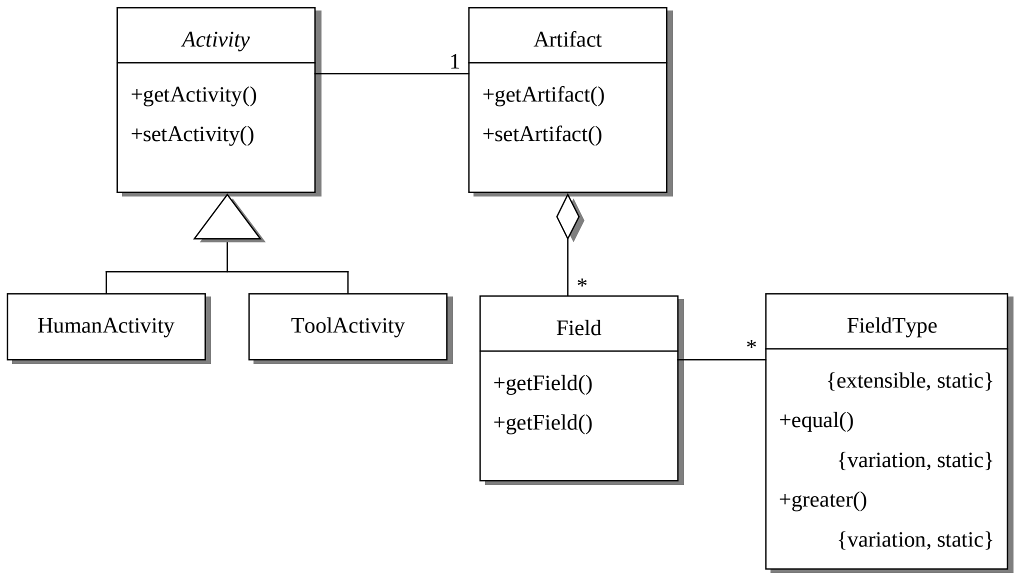


Figure 10.13. Activities, artifacts, and the field type hot-spot

Tools: tools are procedures that update the content of artifacts. Each instance must define its own tools. For each tool defined in the system the method fill must be defined. The tool hot-spot is illustrated in Figure 10.14;

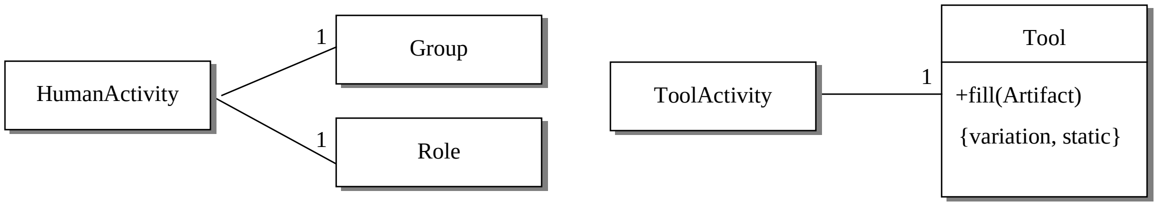


Figure 10.14. Human activities, tool activities, and the tool hot-spot

Exception: exceptions are methods that can modify the configuration of the process state, possibly changing the token positions. Each PROC instance must define its own exceptions, as shown in Figure 10.15;

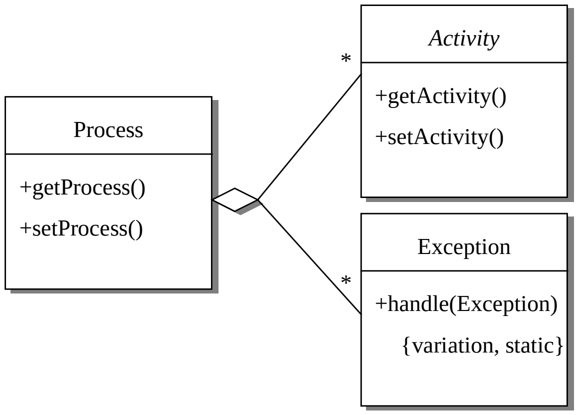


Figure 10.15. Process and the exception hot-spot

Static views: different views of the process descriptions may be created. These views serve as documentation of which processes are defined and how they work;

Dynamic views: different views on the process execution may be created. These views include: listing the process execution trace, reporting the times taken to perform each activity, the average time that actor X executes activity Y, and several other project control reports.

Figure 10.16 presents the system enactor. Each time a process is executed a new *ProcessInstance* object is created. Figure 10.16 also shows the classes that model the static and dynamic hot-spot views.

The extended class diagrams presented so far specify the hot-spots and their semantics.

The next step is to provide implementation solutions for each one of them.

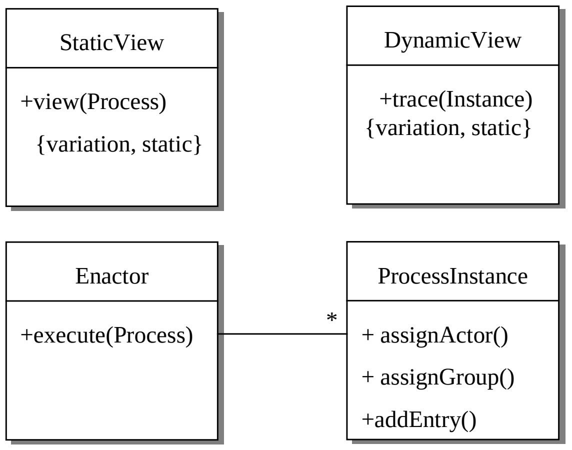


Figure 10.16. Process enactor, static views and dynamic views

The *Role, Group*, and *Actor* hot-spots are variation methods that have a limited behavior. The *get* and *set* functions must vary as their properties vary. Also they need to be dynamically reconfigured. Thus, one possible solution for implementing them is the use of MOP. An *edit()* function that receives a list of properties and stores it internally in the class is defined. When a get or set function is invoked, it reads the list of properties and for each property *p* in the list it invokes *getProperty(p)* or *setProperty(p)*. This solution has been defined in an implementation model and automatically applied to the three hot-spots. The diagram generated is illustrated in Figure 10.17.

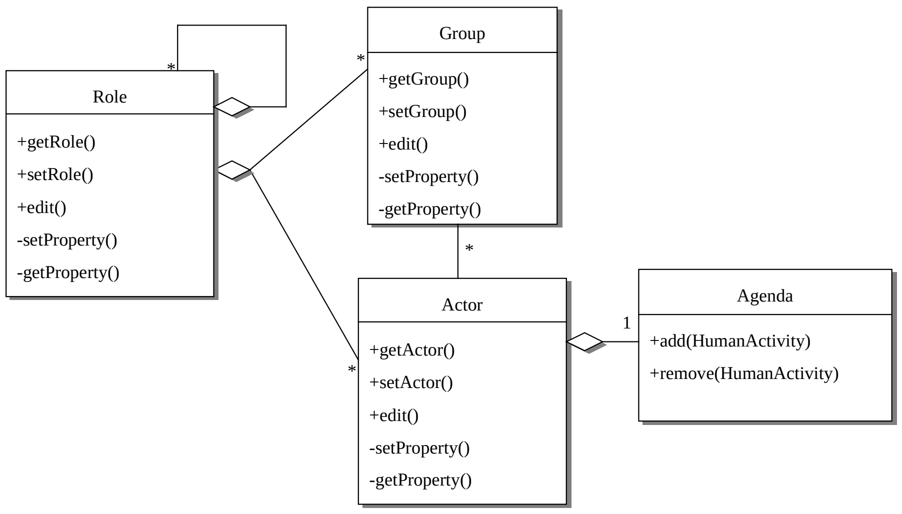


Figure 10.17. Group, role, and actor implementations

The *FieldType* hot-spot is an extensible class since for each *FieldType* several comparison methods can be defined. As modeled, the *equal()* and *greater()* comparisons must be modeled for all types but may be a different implementation for each type. Another point is that the same comparison may be used within several field types. Since several field types may be necessary, the solution chosen for this hot-spot was the use of restricted extension interfaces combined with the visitor pattern, as shown in Figure 10.18. The abstract class *FieldType* provides a default implementation for the equal and *greater() methods()*.

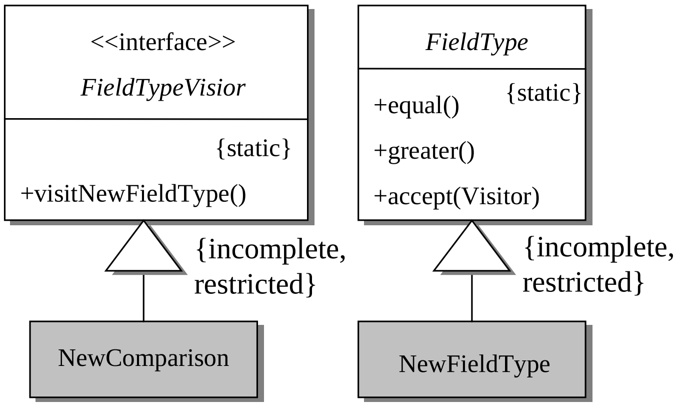


Figure 10.18. FieldType implementation

Note that the visitor pattern could fail in this situation, since when new field types are defined new methods would have to be defined in all the visitor hierarchy. However, since this structure is generated automatically by the implementation model once filed types are defined this problem does not occur. Also, this structure allows new comparisons to be defined in the visitor hierarchy independently of the existing types.

The other four hot-spots have been modeled using restricted interface extensions, as illustrated in Figure 10.19.

Figure 10.20 presents the PROC instantiation diagram. Note that at least one *FieldType* must be defined in the system. The other hot-spots *(Tool*, *Exception*, *DynamicView* and *StaticView)* are optional. *Role*, *Group*, and *Actor* have been eliminated from the instantiation diagram since they have been implemented using MOP and do not require compile-time instantiation.

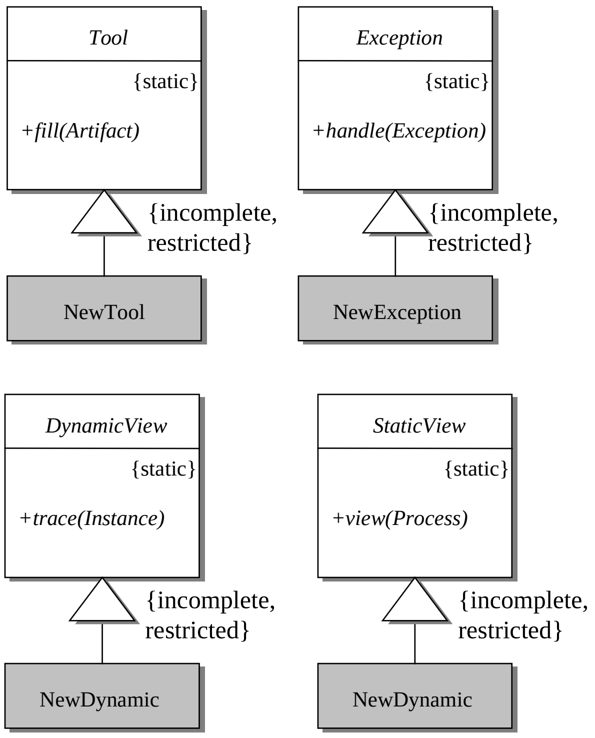


Figure 10.19. Restricted extension interface-based implementation

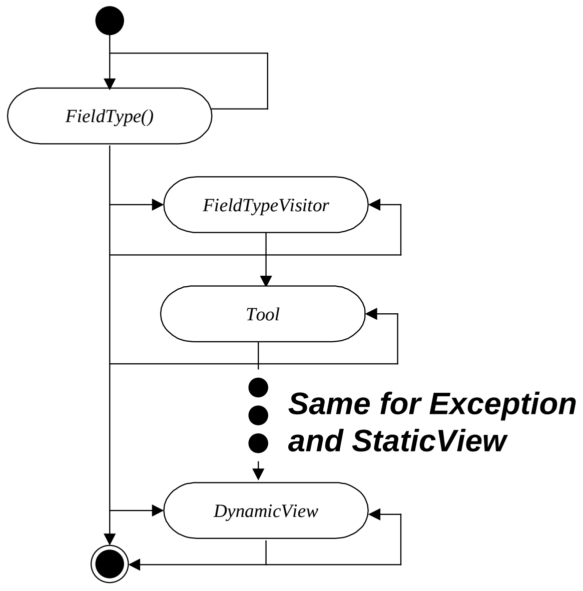


Figure 10.20. PROC instantiation diagram

The PROC hot-spot subsystem was completely generated by the supporting tools proposed in this dissertation, as described in this section. The implementation model that supported the transformations is presented in Figure 10.21. Note that the *generatePROC-MOP* predicate was specifically developed for this project and applies the code defined for the *edit()*, *getProperty()*, and *setProperty()* methods. The PROC kernel is now being developed in Java.

```
implementPROC(Project, NewProject) :-
     (a)copyUMLElements(...),
     (b)forall(variationMethod(..., dynamic), generatePROC-MOP(...)),
     (c)forall(variationMethod(..., static), generateExtensionInt(...)),
     (d)forall(extensionClass(...), visitor(...)),
     (e)forall(transition(...), createTransition(...)),
        [...]
```

(a) Copy the standard kernel elements
(b) Model dynamic variation methods using MOP
(c) Model static variation methods using restricted extension interfaces
(d) Model extension classes using visitor also converts extension class states
(e) Copy instantiation diagram transitions

Figure 10.21. PROC implementation model

One of the intended uses of PROC is the definition of a more sophisticated *process-based instantiation tool*, in which the instantiation diagrams will be modeled as Petri Nets. This instance will have class names, methods names, and method implementations as artifact fields and DSL compilers as tools. This will be a powerful tool for supporting hybrid instantiation (process-based and DLS-based): the human activities will involve defining components to be plugged into the hot-spots while the tool activities will be the invocation of DSL compilers.

10.4 V-MARKET

The framework proposed is greatly inspired on Media Lab's Kasbah [14] system, and concentrates mainly on the ability to create applications based on virtual marketplaces, where buying and selling agents interact. V-Market is a powerful experimentation and research tool, which allows for the fast development of new robust marketplace applications in a fairly simple way [77].

The most important hot-spots in the V-Market architecture are summarized below.

Multiple item support, and structured item ontology: One of the main problems faced with the current implementation of Kasbah [14] is that it is completely tied to the two types of goods that it now supports (books and CDs). To add a new type of product it is necessary to make a major change to the system's structure. The current implementation of the buying and selling agents and the persistence scheme are extremely tied to the specific description of each item.

V-Market addresses these issues by allowing for new types of goods to be easily added to the virtual marketplace. For this to be possible, the goods' definitions should be generic enough to support not only commodity type goods such as books and CDs, but also intangible type of goods such as knowledge about a specific subject, skills, or services. Also, a standard item description/structure must be developed, so that, agents do not need to be redesigned for every new item added to the marketplace.

Each item in the framework must be able to store and compare its attributes. Figure 10.22 illustrates this hot-spot.

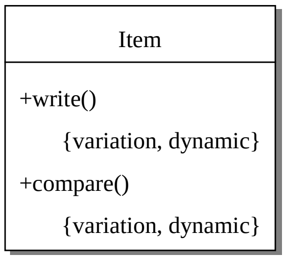
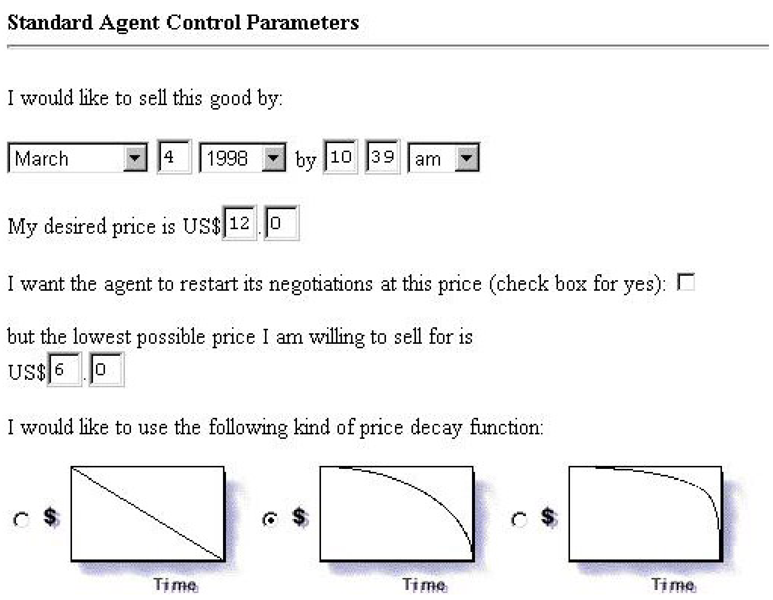


Figure 10.22. Item hot-spot

Multiple negotiation dimensions and strategies: Kasbah [14] allows for only one negotiation dimension (price) and three negotiation strategies for this specific dimension: anxious, greedy/frugal, cool-headed. These negotiation strategies are basically price decay functions over time in the case of selling agent, or price increase functions in the case of buying agents. The anxious strategy varies the price through a linear function, the cool-headed uses a quadratic function, and the greedy/frugal strategies uses a cubic function (Figure 10.23).


Standard Agent Control Parameters

I would like to sell this good by:

March 4 1998 by 10 39 am

My desired price is US$12.0

I want the agent to restart its negotiations at this price (check box for yes): ☐

but the lowest possible price I am willing to sell for is
US$6.0

I would like to use the following kind of price decay function:

○ ● ○

Figure 10.23. Kasbah negotiation strategies

Although these negotiation strategies are quite simple and do not necessarily obtain the best deals in a given virtual marketplace they have been chosen because after some experiments it has been identified that users normally do not fully trust their agents if they do not completely understand what their rationale is. Thus, these strategies have been simplified for the sake of user understanding and trust.

Once these trust barriers are overcome, it will be important to experiment with different and more complex strategies. Some of them might even take market statistics to optimize each deal. For this to be possible, the agent negotiation needs to be specified as a separate entity in the agent component, making it easier to "plug" different negotiation strategies, even at runtime. This type of approach also allows for the creation of negotiation processes for systems that negotiate over more than one dimension[8] as well as to create many different strategies for them.

Multiple communication protocols: Kasbah [14] agent communication protocol is fairly simple. Both buying and selling agents can make buying or selling offers basically composed of a price offer for a specific item and the answer for this offer. The answer

[8] An example of two dimension negotiation may be a negotiation involving price and quality of a book, in a used book marketplace.

can be of two types: positive, in which case the deal is closed, or negative, in which case the agents keep looking for other agents and adapting their prices over time. This type of protocol is well suited for simple negotiation, but for more than one dimension and more complex strategies this protocol would probably not work, or at least it would be very inefficient. It is desirable to support a scalable communication protocol, in which agents could support more complex protocols and in which more elaborated proposals and counter-proposals would be possible.

Figure 10.24 models the last two hot-spots, in which the *doThing(), createProposal() , sendProposal(), processProposal(), and ProcessAnswer()* methods are used to define the communication protocol. Whenever creating a proposal a negotiation strategy must be selected.

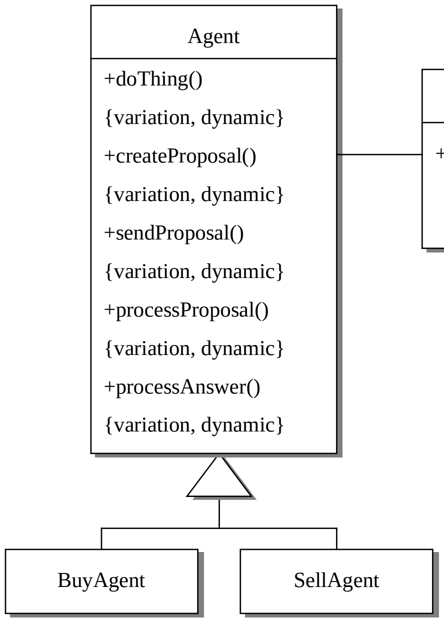


Figure 10.24. Negotiation strategies and communication protocols

Once the hot-spots and their semantics are described precisely they need to be implemented using current OO technology.

The item hot-spot is the most important one and should be instantiated easily. The solution adopted was similar to the one used for the *Group* hot-spot in PROC: an MOP was developed to allow the run-time definition of new items. This solution is shown in Figure 10.25, in which one *Item* is defined as a list of *MetaItem* objects.

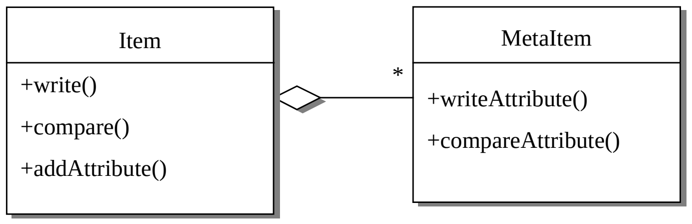


Figure 10.25. Item MOP

To enhance the instantiation of this hot-spot a DSL compiler has been defined. It parses an XML description of the new instances and generates the HTML files that will interface with the end user and creates the new items using the MOP methods. Figure 4 illustrates the DTD used to instantiate the items.

```
<!ELEMENT ITEM (NAME,DESCRIPTION, ATTRIBUTE+)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT DESCRIPTION (#PCDATA)>
<!ELEMENT VALUE (#PCDATA)>
<!ELEMENT LABLE (#PCDATA)>
<!ELEMENT DIRECTIONS (#PCDATA)>
<!ELEMENT ATTRIBUTE (NAME, LABEL, DIRECTIONS?, DESCRIPTION, PRESETS*) >
        <!ATTLIST ATTRIBUTE ATYPE (text|number) "text">
        <!ATTLIST ATTRIBUTE COMPARISON
        (equal|similar|no|numericalBigger|numericalSmaller) #REQUIRED>
        <!ATTLIST ATTRIBUTE INPUT_TYPE
        text|textarea|combobox|radio|checkbox)#REQUIRED>
        <!ATTLIST ATTRIBUTE BROWSEBLE  (yes|no)#REQUIRED>
        <!ATTLIST ATTRIBUTE REQUIRED  (yes|no)#REQUIRED>
        <!ATTLIST ATTRIBUTE SIZE CDATA "45">
<!ELEMENT PRESETS ((VALUE,LABLE)+|(VALUE)+)>
```

Figure 10.26. Item DTD

The other two hot-spots have been instantiated through the strategy and state design patterns as shown in Figure 10.27. New negotiation strategies and the protocol methods are defined in subclasses of S*trategy* and *State*, respectively.

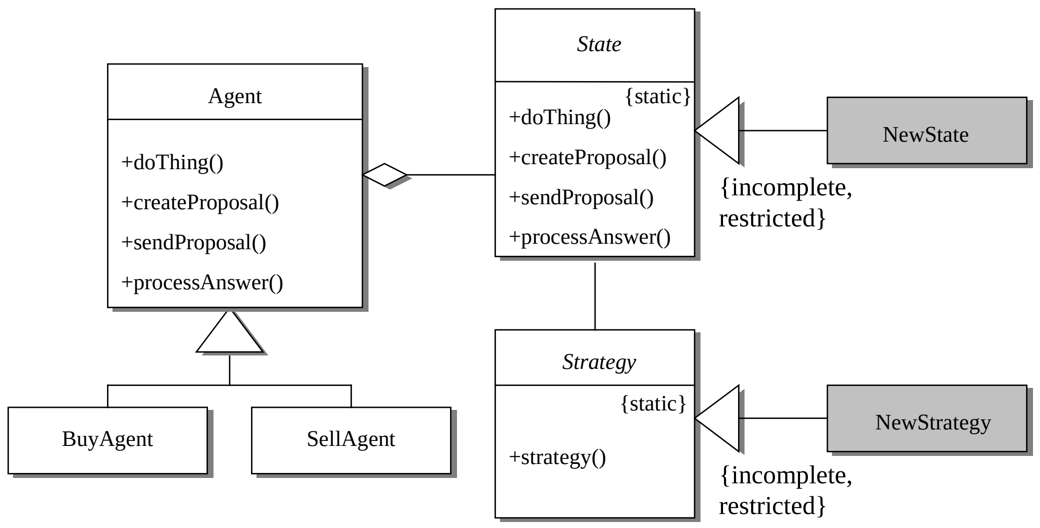


Figure 10.27. Using strategy and state to implement hot-spots

Currently V-Market is completely implemented in Java and a book marketplace instance is being developed to validate the framework.

10.5 ALADIN

This section shows how the ALADIN framework [31] has been developed. The first step in the process was the development of the first version of the AulaNet web-based education environment [21]

Several other web-based educational environments have been studied in order to see if their functionality could be supported by AulaNet [21]. Based on this analysis it has been observed that although many concepts were common to several applications, there were still several concepts that would vary from one application to another.

The idea was then to incorporate all these new ideas into AulaNet and to define a framework that could instantiate all the models that had been analyzed. The list of unifications that have been applied to the AulaNet design structure transforming it gradually into the ALADIN framework is presented next. These transformations have not been applied with tool support since the maintenance tool was not available at the time.

1. Add new kinds of services: services can have different types, such as administrative services (course agenda, bulletin board, etc.), communication services (chat, e-mail, etc.), teaching services (slides, references on the Web, etc.), and assessment services

(quizzes, self-assessment, etc.) but new services may need to be defined. Services can also be internal *(Internal)* or external *(External)*. Internal services are implemented by the environment while the external ones are WWW applications not implemented by the environment, and are provided elsewhere on the Internet. Examples of external services are chat applications, CU-SeeMe, e-mail, and list servers. The addition of new internal and external services, which was supported by the Web-CT (http://homebrew.cs.ubc.ca/webct/) and LearningSpace (http://www. lotus.com/home.nsf/welcome/learnspace) environments, were not supported by ALADIN before this unification;

2. Add new kinds of actors. At the beginning only three kinds of actors were supported: Student, Teacher, and Administrator. However, as the LiveBOOKs [19] environment did support new actor roles this capability was added to ALADIN through unification, as presented in Figure 8.3;
3. New interface capabilities: several environments had support to interface customization, like Web-CT and WCB (http://views.vcu.edu/wcb/intro/wcbintro. hml). This capability was added to the ALADIN framework by unification.
4. New course selection mechanisms: this hot-spot is not supported by any other environment but was added to ALADIN since the selection mechanism had to be changed in several of its instances (Figure 4.16);
5. New student features: Web-CT and Virtual-U (http://virtual-u.cs.sfu.ca/vuweb/) allow for the definition of new features to enhance the student subsystem, such as the TipOfTheDay method. This hot-spot was added through unification as illustrated in Figure 8.2;

After these unifications the design of ALADIN was transformed into its final version. ALADIN can be used to generate each of the analyzed environments as illustrated in [2, 31] (Figure 10.28). It is important to note that the design of each one of the analyzed environments was defined based on the study of its functionality and trying to use the concepts provided by the AulaNet environment. If the designs were completely different from one application to another the unification result would not be so good. In that case, a possible solution is the use of refactoring to adapt the designs before the unification is applied.

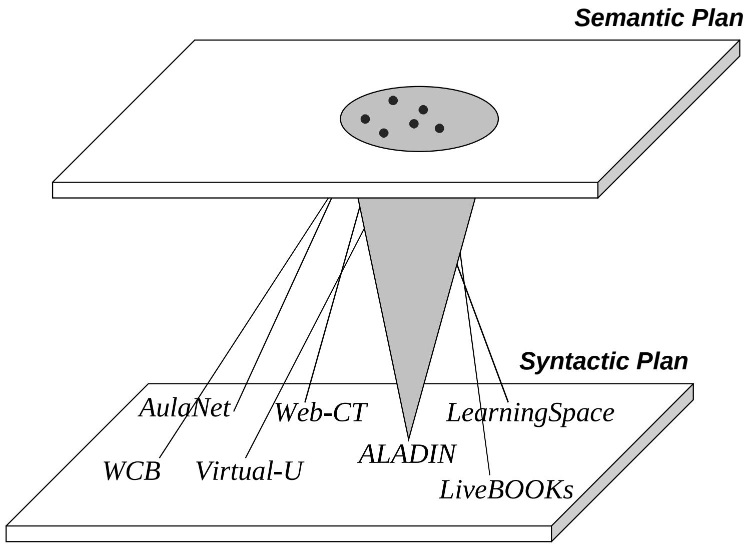


Figure 10.28. Unification-based development process: a formal view

10.6 SUMMARY

Several case studies have been presented to illustrate the language applicability. The Neighbor [5] case study has shown that the use of the language can simplify the design diagrams and highlight hot-spot interdependencies. Also it has shown that the *DSL generator* tool can help the systematization of the framework instantiation by generating a DSL description. The Unidraw [96] framework has shown that the language can give support to frameworks that provide several ways of adding the same functionality. The instantiation diagram in this case has greatly enhanced the understanding of how valid framework instances might be defined. This case study has also shown how the language can be easily adapted to support new specification requirements. The PROC [30] and V-Market [77] case studies have shown how the language and its supporting tools can be used to assist framework development. In these examples it became clear that the framework designer should first focus on what the hot-spots are and on their semantics, leaving the implementation to later steps in the development process. Finally, the ALADIN [31] case study has shown how the unification-based development process can be applied to a real-world example.

Currently, there are three more frameworks being modeled by using the design language with tool assistance: the PJ framework [14], the Quality-of-Service framework [35], and the Multicast framework [80].

Chapter 11 – Conclusions and Future Work

This chapter presents our conclusions based on the work described thorough the dissertation and lists the contributions, describing how they have been accomplished. Finally, it suggests directions for future research.

11.1 CONCLUSIONS

This work presents a design language, which can be used to support systematic framework development. A transformational development process and a set of tools have been defined to illustrate the usefulness of the language. The formalization of the language semantics and the study of the computational complexity of algorithms that verify the transformations applied by the tools are used to show that it can be used as a basis for a rigorous software development process. The language formalization improves its understanding and supports the definition of new development processes and supporting tools.

11.2 LIST OF CONTRIBUTIONS

- Framework design language: the development process and the supporting tools presented in this dissertation confirm the usefulness since most of the systematization is only possible because of the expression power of the underlying framework design language. The case studies show that the language is still useful even if no supporting tools are provided since it reduces design complexity and improves communication between framework designers and application developers.
- Supporting tools: as described throughout the dissertation, the supporting tools proposed in this work can help the systematization of several steps in the development process. Also, the *verifier tool* gives support to the verification of transformations, allowing for a rigorous approach to framework development.
- Domain-specific languages (DSLs) and framework instantiation: as shown in chapters 7 and 10, domain-specific languages may enhance framework instantiation by syntactically securing proper instantiation. The *DSL generator* helps the definition of the *framework structure language.* As far as we know, other approaches for (semi)-

automatically generating DSLs from the framework design have not yet been proposed in the literature.

- Study of several implementation and documentation techniques: the implementation transformations described in chapter 6 (and their formalization presented in chapter 9) describe how several approaches may be used to implement the framework hot-spots. The same is valid for the documentation approaches described in chapter 6.
- Unification rules: the unification rules provide an adequate solution for the architectural drift problem [16]. A unification-based development process may also be used to systematize the derivation of framework abstractions from concrete examples [79].
- Formalization of the concept of frameworks: the formal definition of frameworks proposed in this work is simple an intuitive. Its adoption by the object-oriented community, however, depends on (non technical) factors beyond the control of the authors.

11.3 FUTURE WORK

Several directions for future research can be proposed:

- Requirements elicitation/domain analysis: a systematic procedure for generating specifications in the proposed design language from requirement level artifacts, such as scenarios [42, 43];
- Architectural patterns: the analysis of architectural styles [1, 13, 87] and how they may be used in combination with the proposed design language may also help the job of framework designers;
- Graphical interface tools: the supporting tools described in this dissertation are based on text files and command line interfaces. The construction of graphical interface tools that allow the specification of diagrams through direct manipulation can vastly improve their usability;
- Sophisticated process management tools: currently the *process-based instantiation tool* only supports centralized execution. Also, it does not provide any project management features. The development of tools that allow for distributed framework instantiation, supporting cooperative work and project management controls is a point

that deserves to be investigated. The incorporation of the metrics proposed in [60] is another aspect that can be used to enhance the *process-based instantiation tool*. The use of the PROC framework, described in chapter 10, is a possible approach;

- More unification-based developments: the ALADIN case study [2] is, until now, the only example of unification-based framework development developed within our research group. Several other case studies should be developed to verify the applicability of the approach;
- Debugging tools: visual debugging tools, such as Jinsight (http://www.research.ibm .com/jinsight/) [69], can profit from design level information for building high-level visualizations. If a debugging tool knows that the visitor design pattern was the implementation approach used to model a given hot-spot, a "visitor visualization" may be constructed, as illustrated in Figure 11.1. Debugging diagrams like this allow for the detection of errors at higher levels of abstractions;

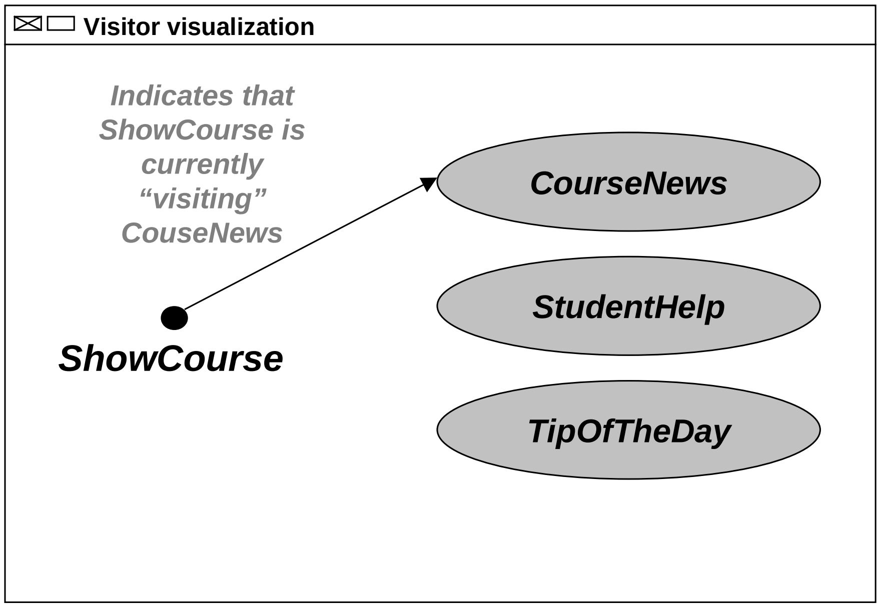


Figure 11.1. The visitor run-time visualization

- Framework composition: integration of more than one framework to create an application might involve several problems as described in [59]. A study of the applicability of the proposed design language to this situation should be investigated. It may be the case that new design level constructs need to be defined to support framework composition more adequately.

- Multi-level frameworks: multi-level frameworks are frameworks that provide multiple interfaces for its users. More experienced framework users may wish to build applications with lower level (and more flexible) interfaces while less experienced users might want to use higher level (and more restrictive) interface frameworks. Figure 11.2 illustrates this approach with an abstract example of a graphical interface framework. The IBM San Francisco (http://www.software.ibm.com/ad/sanfrancisco/) is an example of a multiple-interface framework. It allows more experienced programmers to work with low-level abstractions (at a component level) while less experienced ones can work with high-level business objects. The construction of tools that use the framework design language and support the development of multi-level frameworks can be achieved, and could be useful for experimenting with this approach.

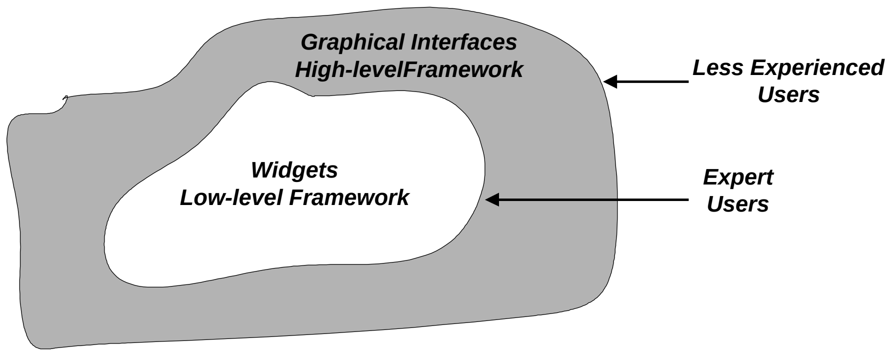
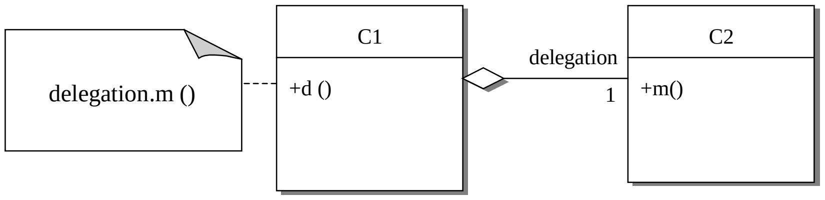


Figure 11.2. Multi-level frameworks

- Language semantics: the design language semantics described in chapter 9 can be extended to incorporate the dynamic aspects of object orientation. This would provide more support for verifying the transformations. Figure 11.3 illustrates two lemmas that may be used to proof of the variation method-strategy transformation (section 9.3.1), the object delegation and the no method lemmas. With these lemmas the proof can be done as follows:

$$\frac{\dfrac{SC(select)}{SC(select), SS()}\ \text{No method}}{SC(selectCourse), SS(select), SC.selectCourse \lozenge SS.select}\ \text{Delegation}$$

New lemmas would have to be defined to support the other transformations.



Delegation Lemma:

$$\frac{C1(M1_1,\ldots, M1_n, m), C2(M2_1, \ldots,M2_k), \Gamma}{C1(M1_1,\ldots, M1_n, d), C2(M2_1,\ldots, M2_k, m),Cl.d \lozenge C2.m, \Gamma}\ \text{delegation}$$

No method Lemma:

$$\frac{\Gamma}{C(\ ), \Gamma}\ \text{no method}$$

Figure 11.3. Object delegation rule

References

1. G. Abowd, R. Allen, and D. Garlan, “Formalizing style to understand descriptions of software architecture”, *ACM Transactions on Software Engineering and Methodology*, 4(4), 319-364, 1995.

2. P. Alencar, D. Cowan, S. Crespo, M. Fontoura, and C. Lucena, “Using Viewpoints to Derive a Conceptual Model for Web-based Education Environments”, Journal of Systems and Software, 1999 (accepted for publication, http://www.les.inf.puc-rio.br/~mafe).

3. P. Alencar, D. Cowan, C. Lucena, and L. Nova, “A Model for Gluing Components”, 3rd International Workshop on Component-Oriented Programming, TUCS General Publication n.10, 101-108, 1998.

4. P. Alencar, D. Cowan, J. Dong, and C. Lucena, “A Transformational Process-Based Formal Approach to Object-Oriented Design”, Formal Methods Europe (FME’97), 1997 (http://csgwww.uwaterloo.ca).

5. A. Andreatta, S. Carvalho, and C. Ribeiro, “An Object-Oriented Framework for Local Search Heuristics”, 26th TOOLS, *IEEE Press*, 33-45, 1998.

6. D. Batory and S. O’Malley, “The Design and Implementation of Hierarchical Software Systems with Reusable Components”, *ACM Transactions on Software Engineering and Methodology*, 1(4), 355-398, 1992.

7. D. Baumer, D. Riehle, W. Siberski, and M. Wulf, “Role Object”, PloP’97, Technical Report WUCS-97-34, Washington University, Department of Computer Science, 1997.

8. D. Bellin and S. Simone, *The CRC Card Book*, Addison Wesley Longman, 1997.

9. J. Bicarregui, K. Lano, and T. Maibaum, “Objects, Associations, and Subsystems: a Hierarchical Approach to Encapsulation”, ECOOP’97, *LNCS 1241*, 324-343, 1997.

10. Borland Inc., *Delphi User’s Guide*, 1995.

11. J. Bosch and Y. Dittrich, “Domain-Specific Languages for a Changing World”, (http://bilbo.ide.hk-r.se: 8080/~bosch/ articles.html).

12. F. Budinsky, M. Finnie, J. Vlissides, and P. Yu, “Automatic Code Generation from Design Patterns”, *Object Technology*, 35(2), 1996.

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

14. S. Carvalho, A. Lerner, and S. Lifschitz, “An Object-Oriented Framework for the Parallel Join Operation”, 10th International Conference and Workshop on Database and Expert Systems Applications (DEXA'99), *IEEE Press*, 1999 (to appear).

15. Chavez and P. Maes. “Kasbah: An Agent Marketplace for Buying and Selling Goods”. *Proceedings of* the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'96), London, UK, April 1996.

16. W. Codenie, K. Hondt, P. Steyaert, and A. Vercammen, “From Custom Applications to Domain-Specific Frameworks”, *Communications of the ACM*, 40(10), 71-77, 1997.

17. J. Coplien, “Broadening beyond objects to patterns and other paradigms”, *ACM Computing Surveys*, 28(4es), 152-es, 1996.

18. J. Cordy and I. Carmichael, “The TXL Programming Language Syntax and Informal Semantics”, Technical Report, Queen’s University at Kingston, Canada, 1993. (http://www.qucis.queensu.ca/STLab/TXL).

19. D. Cowan , An Object-Oriented Framework for LiveBOOKs, Technical Report, CS-98, University of Waterloo, Ontario, Canada, 1998.

20. D. Cowan and C. Lucena, “Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse”, *IEEE Transactions on Software Engineering*, 21(3), 229-243, 1995.

21. S. Crespo, M. Fontoura, and C. Lucena, “AulaNet: An Object-Oriented Environment for Web-based Education”, ICLS’98, *AACE Press*, 304-306, 1998.

22. S. Crespo, M. Fontoura, and C. Lucena, “Object-Oriented Design Course”, Computer Science Department, PUC-Rio, http://ead.les.inf.puc-rio.br/aulanet (in Portuguese).

23. D. D’Souza and A. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison Wesley, 1997.

24. A. Eden, J. Gil, and A. Yehudai, “Precise Specification and Automatic Application of Design Patterns”, ASE’97, *IEEE Press,* 1997.

25. A. Eden, Y. Hirshfeld, and A. Yehudai, “LePUS - A Declarative Pattern Specification Language”. Technical Report 326/98, Department of Computer Science, Schriber School of Mathematics, Tel Aviv University, 1998.

26. A. Eden, Y. Hirshfeld, and A. Yehudai, “Precise Notation for Design Patterns. Technical Report 327/98, Department of Computer Science, Schriber School of Mathematics, Tel Aviv University, 1998.

27. M. Fayad and D. Schmidt, “Object-Oriented Application Frameworks”, *Communications of the ACM*, 40(10), 32-38, 1997.

28. J. Fiadeiro and T. Maibaum, “Describing, Structuring, and Implementing Objects”, *LNCS 489,* Springer-Verlag, 274-310, 1991.

29. G. Florijin, M. Meijers, P. van Winsen, “Tool Support for Object-Oriented Patterns”, ECOOP’97, *LNCS 1241*, Springer-Verlag, 472-495, 1997.

30. M. Fontoura, “An Environment for Process Modeling and Execution”, M.Sc. Dissertation, Computer Science Department, PUC-Rio, 1997 (in Portuguese).

31. M. Fontoura, L. Moura, S. Crespo, and C. Lucena, “ALADIN: An Architecture for Learningware Applications Design and Instantiation”, Technical Report MCC34/98, Computer Science Department, PUC-Rio, 1998 (also submitted to WWW Journal).

32. G. Froehlich, H. Hoover, L. Liu, and P. Sorenson, “Hooking into Object-Oriented Application Frameworks”, ICSE’97, *IEEE Press*, 491-501, 1997.

33. G. Froehlich, H. Hoover, L. Liu, and P. Sorenson, “Requirements for a Hooks Tool”, (http://www.cs.ualberta.ca/~softeng/papers/papers.htm).

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

35. A. Gomes, S. Colcher, L. Soares, “A Framework for QoS in Generic Environments of Processing and Communication”, Brazilian Symposium of Computer Networks’1998, *SBC,* 2998.

36. M. Griss, “Systematic Software Reuse: Objects and Frameworks are Not Enough”, *Object Magazine,* February, 1995.

37. M. Griss and K. Wentzel, “Hybrid Domain-Specific Kits”, *Journal of Systems and Software,* February, 1995

38. W. Harrison and H. Ossher, “Subject-Oriented Programming (A Critique of Pure Objects)”, OOPSLA’93, *ACM Press*, 411-428, 1993.

39. R. Helm, I. Holland, and D. Gangopadhyay, “Contracts: Specifying Behavioral Composition in Object-Oriented Systems”, OOPSLA/ECOOP’98, Norman Meyrowitz (ed.), *ACM Press*, 169-180, 1990.

40. I. Holland, “The Design and Representation of Object-Oriented Components”, Ph.D. Dissertation, Computer Science Department, Northeastern University, 1993.

41. P. Hudak, “Building Domain-Specific Embedded Languages”, *ACM Computing Surveys*, 28(4es), 196-es, 1996.

42. I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering,* Wokingham: Addison-Wesley/ACM Press, 1993.

43. I. Jacobson, M. Ericsson, and A. Jacobson,. *The Object Advantage,* Wokingham: Addison-Wesley/ACM Press, 1995.

44. S. Jarzabeck and T. Ling, “Knowledge Model for Business Process Re-engineering Tools”, TRC3/1995, Department of Information Systems and Computer Science, National University of Singapore, 1995.

45. R. Johnson, “ Documenting Frameworks Using Patterns”, OOPSLA’92, *ACM Press,* 63-76, 1992.

46. R. Johnson, “Frameworks = (Components + Patterns)”, *Communications of the ACM*, 40(10), 39-42, 1997.

47. R. Johnson and W. Opdyke, “Refactoring and aggregation”, Object Technologies for Advanced Software, First JSSST International Symposium, *LNCS 742*, 264-278, 1993.

48. W. Johnson and E. Soloway, “PROUST: Knowledge-Based Program Understanding”, Conference on Software Maintenance’1985, *IEEE Press*, 369-380, 1985.

49. M. Kaplan, H. Ossher, W. Harrison, and V. Kruskal, “Subject-Oriented Design and the Watson Subject Compiler”, OOPSLA’96 Subjectivity Workshop, 1996 (http://www.research.ibm.com/sop/).

50. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, “Aspect-Oriented Programming”, ECOOP’96, *LNCS 1241*, 220-242, 1997.

51. G. Kiczales, J. des Rivieres, and D. Bobrow, *The Art of Meta-object Protocol*, MIT Press, 1991.

52. G. Krasnes and S. Pope, “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80”, *Journal of Object-Oriented Programming*, 1(3), 26-49, 1988.

53. K. Lano, “Logical Specification of Reactive and Real-time Systems”, *Journal of Logic and Computation*, 8(5), 679-711, 1998.

54. J. Leite, M. Sant'Anna, and A. do Prado, “Porting Cobol Programs Using a Transformational Approach”, *Journal of Software Maintenance*, John Wiley & Sons, 9, 3-31, 1997.

55. S. Letovsky and S. Soloway, “Delocalized Plans and Program Understanding”, *IEEE Software*, 3(3), 41-49, 1986.

56. K. Lieberherr and I. Holland, “ Assuring Good Style for Object-Oriented Programs”, *IEEE Software*, September, 38-48, 1989.

57. C. Lopes and G. Kiczales, “Recent Developments in AspectJ”, ECOOP'98 Workshop Reader, 1998 (http://www.parc.xerox.com/spl/groups/eca/pubs/).

58. Manna and R. Waldinger, “Fundamentals of Deductive Program Synthesis”, *IEEE Transactions on Software Engineering*, 18(8), 674-704, 1992.

59. M. Mattsson, “Object-Oriented Frameworks: A Survey of Methodological Issues”, M.Sc. Dissertation, Department of Computer Science and Business Administration, University College of Karlskrona/Ronneby, LU-CS-96-197, 1996.

60. M. Mattsson, and J. Bosch, “Assessing Object-Oriented Application Framework Maturity – A Replicated Case Study”, to appear (http://bilbo.ide.hk-r.se: 8080/~bosch/ articles.html).

61. T. Meijler, S. Demeyer, and R. Engel, “Making Design Patterns Explicit in FACE – A Framework Adaptative Composition Environment”, ESEC’97, *LNCS 1301*, Springer-Verlag, 94-111, 1997.

62. M. Mezini and K. Lieberherr, “Adaptative Plug-and-Play Components for Evolutionary Software Development”, OOPSLA’98, *ACM Press*, 97-116, 1998.

63. Microsoft Inc., *Microsoft Visual C++ 4.0 User’s Guide*, 1995.

64. J. Neighbors, “The Draco Approach to Constructing Software from Reusable Components”, *IEEE Transactions on Software Engineering*, 10(5), 564-574, 1984.

65. OMG, “OMG Unified Modeling Language Specification V.1.2”, 1998 (http://www.rational.com/uml).

66. W. Opdyke, “Refactoring Object-Oriented Frameworks”, Ph.D. Dissertation, Computer Science Department, University of Illinois, Urbana-Champaign, 1992.

67. P. Pal, “Law-Governed Support for Realizing Design Patterns”, 17th TOOLS, *IEEE Press*, 1995.

68. D. Parnas, P. Clements, and D. Weiss, “The Modular Structure of Complex Systems”, *IEEE Transactions on Software Engineering*, SE-11, 259-266, 1985.

69. W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides, “Visualizing the behavior of object-oriented systems”, OOPSLA’93, *ACM Press*, 326-337, 1993.

70. W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

71. W. Pree, *Framework Patterns*, Sigs Management Briefings, 1996.

72. R. Prieto-Diaz and G. Arango (eds.), *Domain Analysis: Acquisition of Reusable Information for Software Construction*, IEEE Press, 1989.

73. T. Pressburger and M. Lowry, "Automatic Domain-Oriented Software Design using Formal Methods", Software Systems in Engineering, Energy-Sources Technology Conference and Exhibition, 33-42, 1995.

74. T. Reenskaug, P. Wold, and O. Lehne, *Working with objects,* Manning, 1996.

75. D. Riehle and T. Gross, "Role Model Based Framework Design and Integration", OOPSLA'98, *ACM Press*, 117-133, 1998.

76. A. Riel, *Object-Oriented Design Heuristics,* Addison-Wesley, 1995.

77. P. Ripper, "V-Market: A Framework for Agent Mediated E-Commerce Systems based on Virtual Marketplaces", M.Sc. Dissertation, Computer Science Department, PUC-Rio, 1999.

78. D. Roberts, J. Brant, and R. Johnson, "A Refactoring Tool for Smalltalk", University of Illinois at Urbana-Champaign, Department of Computer Science (http://st-www.cs.uiuc.edu/users/droberts/).

79. D. Roberts and R. Johnson, "Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks", in *Pattern Languages of Program Design 3,* Addison-Wesley, R. Martin, D. Riehle, and F. Buschmann (eds.), 471-486, 1997.

80. M. Rodrigues, S. Colcher, L. Soares, "A Framework for Multicast in Generic Environments of Processing and Communication", Brazilian Symposium of Computer Networks'1998, *SBC,* 2998.

81. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Clifs, 1991.

82. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998.

83. M. Sant'Anna, "Transformational Circuits", Ph.D. Dissertation, Computer Science Department, PUC-Rio, 1999 (in Portuguese).

84. M. Sant'anna, J. Leite, A. do Prado, “Draco-PUC: A Workbench For Developing Transformation-Based Software Generators”, ICSE'98, *IEEE Press*, vol. 2, 135-139, 1998.

85. Marcelo Sant'Anna, Julio Leite, Antonio do Prado, “A Generative Approach to Componentware”, CBSE'98, International Workshop on Component Based Software Engineering, 1998 (http://www.sei.cmu.edu/activities/cbs/icse98/papers/).

86. R. Seviora, “Knowledge-Based Program Debugging Systems”, *IEEE Software,*4(3), 20-32, 1987.

87. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline,* Prentice Hall, 1996.

88. H. Schmid, “Systematic Framework Design by Generalization”, *Communications of the ACM*, 40(10), 48-51, 1997.

89. R. Silva e R. Tom Price, “O Uso de Tecnicas de Modelagem no Projeto de Frameworks Orientados a Objetos”, Argentine Symposium on Object Orientation, *SADIO,* 89-94, 1997 (in Portuguese).

90. D. Smith, “KIDS-A Knowledge-Based Software Development System,” in *Automating Software Design*, M. Lowry and R. McCartney, (eds.), AAAI/MIT Press, 483-514, 1991.

91. D. Smith and T. Pressburger, “Knowledge-Based Software Development Tools”, in *Software Engineering Environments*, P. Brereton (ed.), Ellis Horwood Ltd., Chichester, 79-103, 1988.

92. M. Snoeck and G. Dedene, “Existence Dependency: The Key to Semantic Integrity Between Structural and Behavioral Aspects of Object Types”, *IEEE Transactions on Software Engineering*, 24(4), 233-251, 1998.

93. M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwook, “Deductive Composition of Astronomical Software from Subroutine Libraries”, 12th Conference on Automated Deduction, 1994.

94. K. Sullivan, J. Socha, and M. Marchukov, “Using Formal Methods to Reason about Architectural Standards”, ICSE’97, *IEEE Press*, 503-513, 1997.

95. J. Vlissides, *Pattern Hatching: Design Patterns Applied*, Software Patterns Series, Addison-Wesley, 1998.

96. J. Vlissides, “Generalized Graphical Object Editing”, Ph.D. Dissertation, Department of Electrical Engineering, Stanford University, 1990.

97. M. Ward, F. Calliss, and M. Munro, “The Use of Transformations in ‘The Maintainer’s Assistant’”, Technical Report 88/9, Computer Science Department, University of Durham, England, 1988.

98. R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

Appendix A – Design Language Summary

This appendix lists the elements of the UML-extended design language. There are five types of diagrams supported by the language: standard class diagrams, extended class diagrams, interaction diagrams, template diagrams, and instantiation. The syntax of each new design element is presented below.

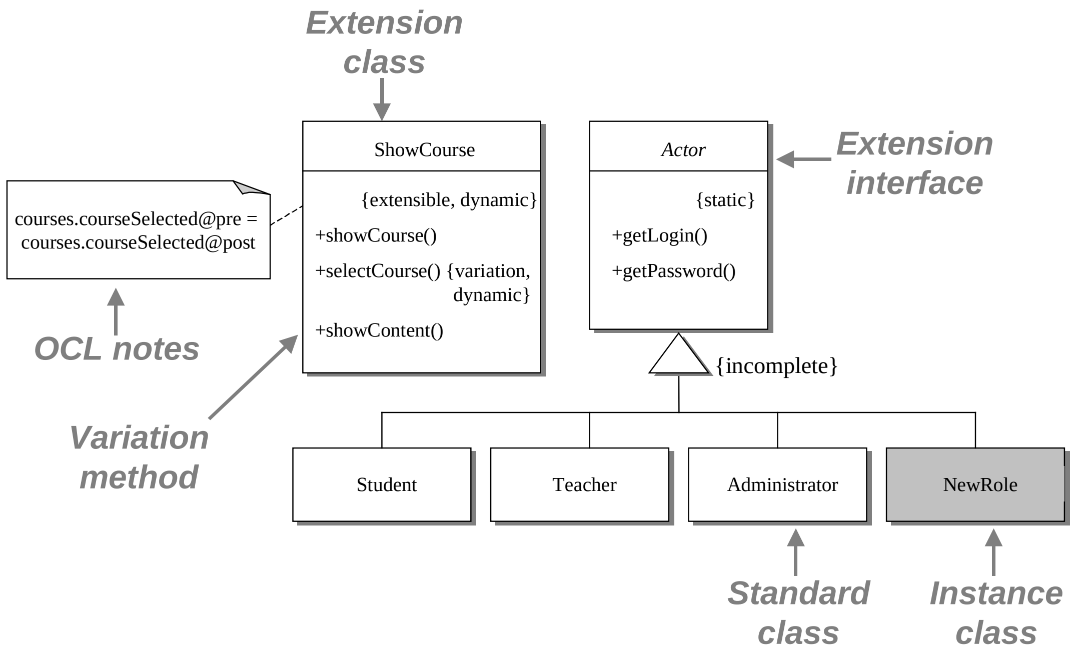


Figure A.1. Extended class diagrams

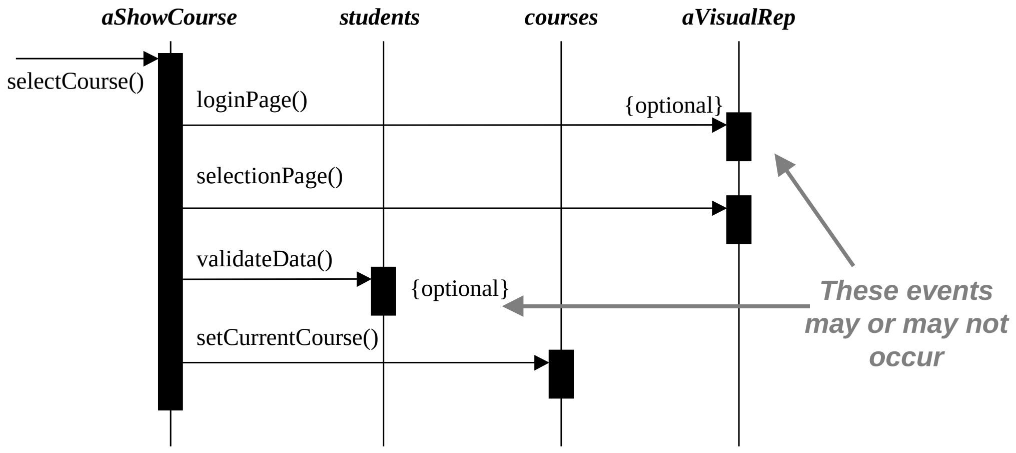


Figure A.2.Template diagrams

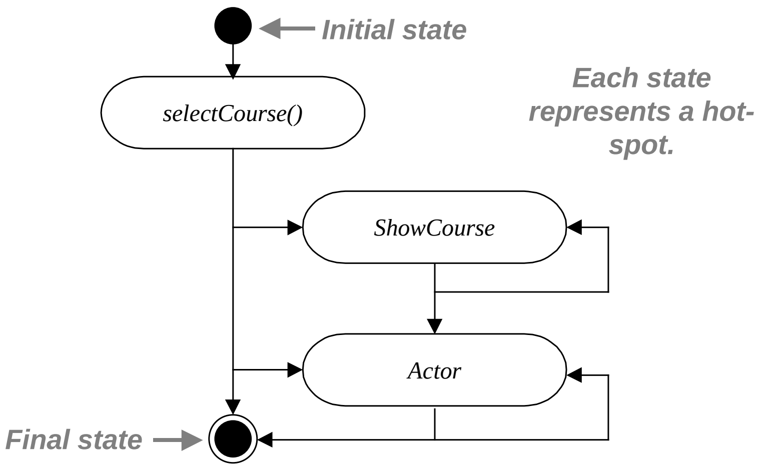


Figure A.3. Instantiation diagrams

Appendix B – Prototype Implementation

The PROLOG tools described throughout the dissertation exemplify how the proposed framework design language may be used to automate several steps of a given framework development process. The tools are in the public domain and may be downloaded from the PROOF project web-site (http://www.les.inf.puc-rio.br/~mafe/proof.html). PROOF is a project for the systematic PRoduction of Object-Oriented Frameworks, developed at the Software Engineering Lab of the Pontifical Catholic University of Rio de Janeiro (LES PUC-Rio). The files that model the several frameworks described in this dissertation are also available for download. The PROLOG currently being used is the SWI-Prolog, developed at the University of Amsterdam. This appendix describes the PROLOG files that compose the tools.

File	Description
Main.pl	Calls all the other files. It also provides the *loadProject* predicate, which is responsible for reading project files.
Uml.pl	Provides transformations for the UML standard class diagrams and interaction diagrams.
Frameworks.pl	Provides transformations for the extended class diagrams, template diagrams, and instantiation diagrams.
Listing.pl	Built-in design analysis that lists project definitions.
Log.pl	Lists the history of transformations.
Copy.pl	Support file that provides several functions for copying elements from one project to another.
Process.pl	Implements the *process-based instantiation tool.*
Dsl.pl	Implements the *DSL generator tool.*
Verifier.pl	Implements the *Verifier tool.*
Analysis.pl	Meta-artifact in which new design analyses may be defined.
Implementation.pl	Meta-artifact in which new implementation models may be defined.
Documentation.pl	Meta-artifact in which new documentation models may be defined.
Refactoring.pl	Meta-artifact in which new refactoring rules may be defined.

Unification.pl	Meta-artifact in which new unification rules may be defined.

Table B.1. List of the PROLOG files

The files that model framework examples are the following:

File	Description
Aladin.pl	Models the ALADIN framework.
Neighbor.pl	Models the Neighbor framework.
Unidraw.pl	Models the Unidraw framework.
Proc.pl	Models the PROC framework.
Vmarket.pl	Models the V-Market framework.
Pj.pl	Models the PJ framework.
Qos.pl	Models the QoS framework.
Multicast.pl	Models the Multicast framework.

Table B.2. List of the example files