A systematic approach for framework development

Marcus Fontoura **Edward H. Haeusler** **Carlos J. Lucena**

Computer Science Department
Pontifical Catholic University of Rio de Janeiro
R. Marquês de S. Vicente, 225
Rio de Janeiro 22453-900
e-mail: {mafe, hermann, lucena}@inf.puc-rio.br

PUC-Rio.Inf.MCC 26/99 Dezembro, 1999

ABSTRACT

This paper presents several object-oriented design techniques and discusses their ability to represent frameworks adequately. Based on the analysis of these techniques requirements for a design representation for frameworks are proposed. A design language based on extensions to UML is introduced to satisfy the requirements by representing frameworks variation points and their instantiation process as first-class citizens. These extensions are defined using the UML extensibility mechanisms: stereotypes, tagged values, and constraints. The use of the language to systematize framework implementation is discussed. An analysis of the design language is presented through two large case studies that show how it can be used to assist framework development and instantiation. The semantic of the new constructors is formally described to allow the verification of the development and instantiation activities.

Key Words: Object-Oriented Design, Frameworks, Design Patterns, UML, Framework Design Language.

1. Introduction

Object-oriented frameworks and product line architectures have become popular in the software industry during the 1990s. A vast number of frameworks has been developed in the industry for various domains, including graphical user interfaces (e.g. Java's Swing, Microsoft's MFC), graph-based editors (HotDraw, Stingray's Objective Views), business applications (IBM's San Francisco), electronic commerce (Sun/IBM), network servers (Java's Jeeves), just to mention few. When combined with components, frameworks provide the most promising current technology supporting large scale reuse [14].

A framework is a collection of several fully or partially realized components with predefined cooperation patterns between them. A framework implements the software architecture for a family of applications [18], to be specialized by application-specific code. Hence, some of these components, which are the variation points or hot-spots [19], are designed to be replaceable. Applications built on top of a framework not only reuse its source code but also its architecture design. This amounts to a standardization of the application structure and implies a significant reduction in the size and complexity of the source code that has to be written by the programmer who instantiates a framework.

The bad news is that framework development requires an extra effort. The costs of identifying the adequate abstractions and structuring the variation points are high. Framework usability is also a problem. It is not always easy to identify which code is needed to instantiate a framework and where it should be written since 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 classes and write the required instantiation code if no adequate documentation is provided.

The main causes of these problems is the lack of proper techniques for describing frameworks, especially the points where application developers need to instantiate the framework. Tools and methods to assist framework development and instantiation are almost non-existent or in their infancy.

The recent standardization efforts of the Unified Modeling Language (UML) [26] provide a unique chance to harness UML as notational basis for framework development projects. This paper presents how to explicitly represent the variation points of frameworks in UML using static and dynamic description diagrams to capture allowed structure and semantics of the variation points. For this purpose, a new member of the UML language family is proposed, including a few extensions to assist framework development. The new extensions have been defined mainly by applying the UML extensibility mechanisms: stereotypes, tagged values, and constraints.

Several projects have already been developed using the UML extended design language proposed in this paper [1, 8, 10, 24]. These experiences have shown how it can assist the framework development and instantiation activities reducing development costs. This paper presents two comprehensive real-world case studies to illustrate the language usability and how it can assist the framework developers and users.

The semantics of the framework design language is present through the use of set theory. 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 supporting tools. The formalization is also used to verify development and instantiation activities.

The rest of this paper is organized as follows: Section 2 describes and analyzes the design techniques currently used to model frameworks. Section 3 outlines the requirements for an adequate framework design language, describes the UML extensions, and shows how they satisfy the requirements. Section 4 describes how the language can be used to assist framework development and instantiation. Section 5 describes case studies in different domains. Section 6 presents the language formal semantics and shows how the verification of the development and instantiation activities can be performed. Section 7 presents our conclusions and future research directions.

2. Current approaches for framework development

This section describes some of the current design techniques used to model frameworks, showing that they do not provide constructs to represent framework variation points adequately.

2.1 Standard OOADMs

Standard OO design methods, like OMT [25] and UML [26], provide static (class and object diagrams) and dynamic (activity, interaction, state-chart, and state diagrams) views of a system design. Figure 1 is a representation of the student subsystem of a web-based education framework [10] 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.

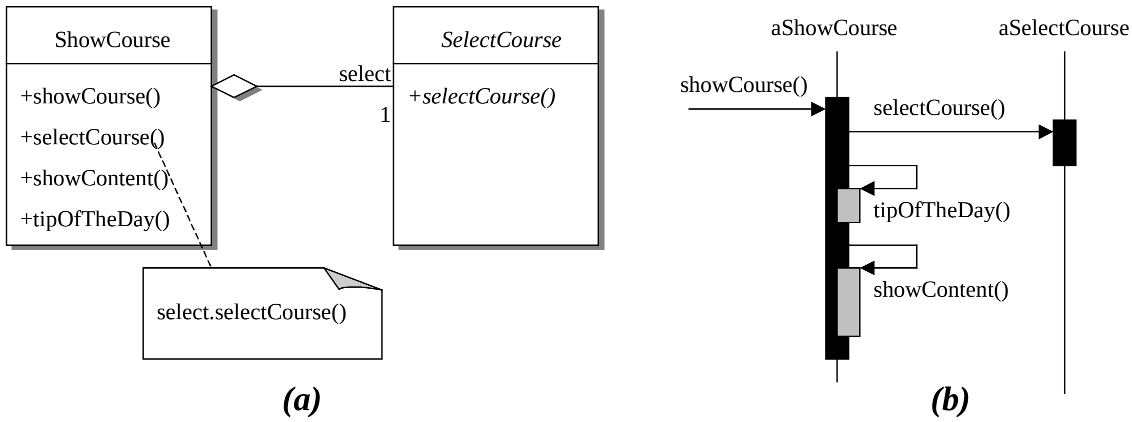


Figure 1. UML representation of a framework

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 variation point 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 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 it. The problem with this representation is that there is no indication in the design diagrams that this method is a variation point. There is also no indication of how it should be instantiated. Although the name of the abstract method *selectCourse()* is italicized, this notation is not an indication of a variation point, rather it indicates an abstract method. Abstract methods do not necessarily represent framework variation points.

The method *tipOfTheDay()* is also a variation point. 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 needed by all possible instantiated applications. The extra methods should only be added to the framework instances. The inclusion of these methods to framework classes could result in a complex interface for those classes, which is a bad design heuristic [23]. Changing the implementation of an existing class every time new functionality is needed is also not good practice. A notation to designate where new 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 variation points from kernel elements. Different OO design notations include different artifacts, such as the representation of object responsibilities as CRC cards [2, 29]. However none of these artifacts are helpful for explicitly representing variation points and how they should be instantiated.

2.2 Considerations about design patterns and meta-level programming

Design patterns [3, 13, 28] are usually described using standard OO notations such as OMT and UML. However, since various design patterns provide solutions to variability and extensibility problems [13] they define a common vocabulary to talk about these concepts [28] and may enhance the understanding of framework designs.

Figure 2 is an example of the use of design patterns to model the variation points of a framework. It uses the Strategy design pattern [13] to model *selectCourse()* and the Visitor design pattern [13] to model *tipOfTheDay()*. 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.

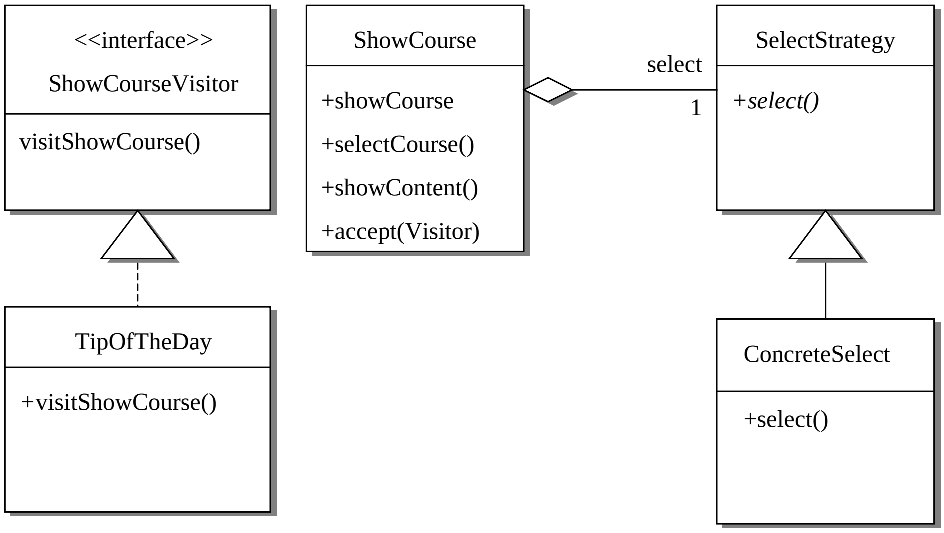


Figure 2. Using design patterns to model frameworks

In this example the design pattern names are used in the class names. This approach can help the framework user in finding the variation points, but only when he or she knows the design patterns that were used for modeling each of these points. In a typical framework design a single variation point class can participate in various design patterns. Then the approach of using design pattern names as class names fails[1]. In addition, representing design patterns with standard OO design notations can lead to very complex designs, even for simple frameworks [9].

Meta-level programming [15], which can be seen as an architectural pattern [3], 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 variation points are expressed through meta-object protocols (MOPs) they can be more easily identified in the design model.

Figure 3 shows an example of the use of meta-programming to model *selectCourse()*. Class *SelectMOP* implements 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 variation points that require run-time instantiation. 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

[1] This is not a problem of the design patterns themselves, but of how they are represented at design level. The role-based modeling technique [22], for example, can be used to solve this problem as described in section 2.3.

allowed by the MOP. In the Strategy approach any 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 variation points, 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.

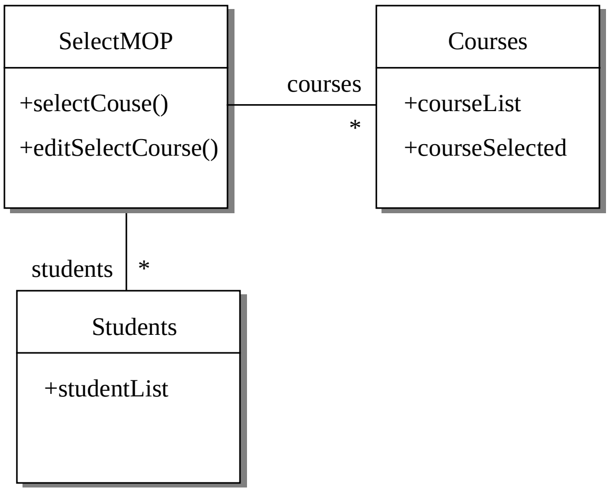


Figure 3. Meta-programming example

2.3 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 [5]. Riehle proposes an extension of the OOram methodology [21] to facilitate framework design and documentation [22]. His work proposes a solution for an explicit division of the design, highlighting the interaction of the framework with its clients.

Figure 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 [13], choosing an appropriate course selection mechanism that is then implemented by subclasses of *SelectCourse*. *ShowCourse* is also an Observer [13] 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 variation point elements. This problem is handled using design patterns in the same way as proposed in section 2.2. If the framework user knows what patterns were used to model each of the variation points 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 variation points becomes very hard.

Another disadvantage of this approach is the solution for modeling unforeseen extensions of framework proposed in [22], which 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.

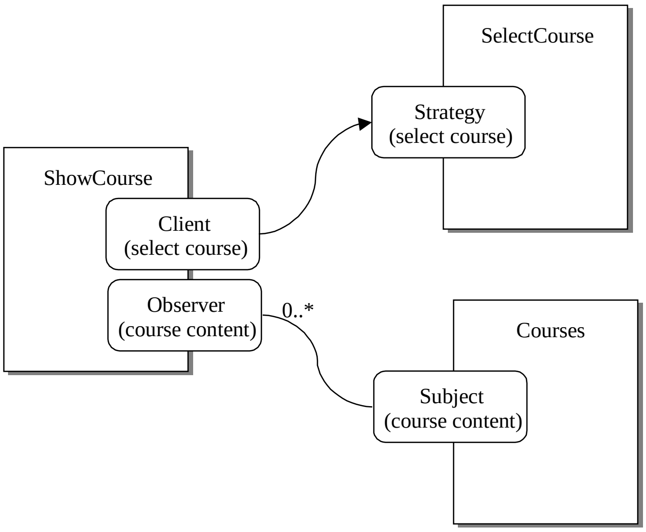


Figure 4. A role-based modeling example

2.4 Catalysis and the UML binding stereotype

UML represents design patterns as collaborations (or mechanisms) and provides a way of modeling framework instantiation through the binding stereotype [26]. Figure 5 shows how *selectCourse()* instantiation may be represented in UML, where *Course Selection* represents the framework subsystem that models it, *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: variation points might have interdependencies, might be optional, and so on. Catalysis follows the UML approach and proposes a design method for building frameworks [6], but still does not solve these problems since the instantiation process is not represented as a first-class citizen.

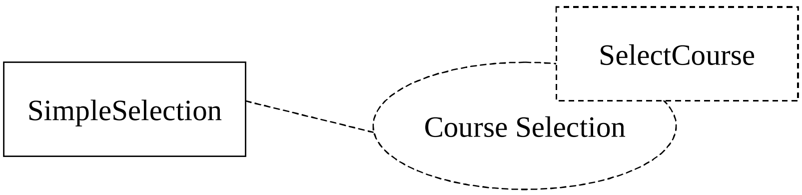


Figure 5. Representing collaborations in UML

2.5 Hot-spot cards and meta-patterns

Pree recognizes that standard OO design methods neglect the identification of variation points, and proposes the use of hot-spot cards as a means for documenting the information required for developing frameworks [19, 20]. The cards document the semantics of each variation point, and are divided into two categories: data and function. Since hot-spot cards are meant to cover the gap between the framework developers and domain experts (who are generally not software engineers), technical terms, such as classes and objects, are avoided.

Figure 6 presents the data and function cards for *selectCourse()*. The "degree of flexibility" field in the function card is used to model the run-time requirements. If "adaptation without restart" is needed the variation point may need to be instantiated during run-time, and if "adaptation by end user" is required it is necessary to build an extra tool to assist with the instantiation process. The remaining fields serve as informal documentation.

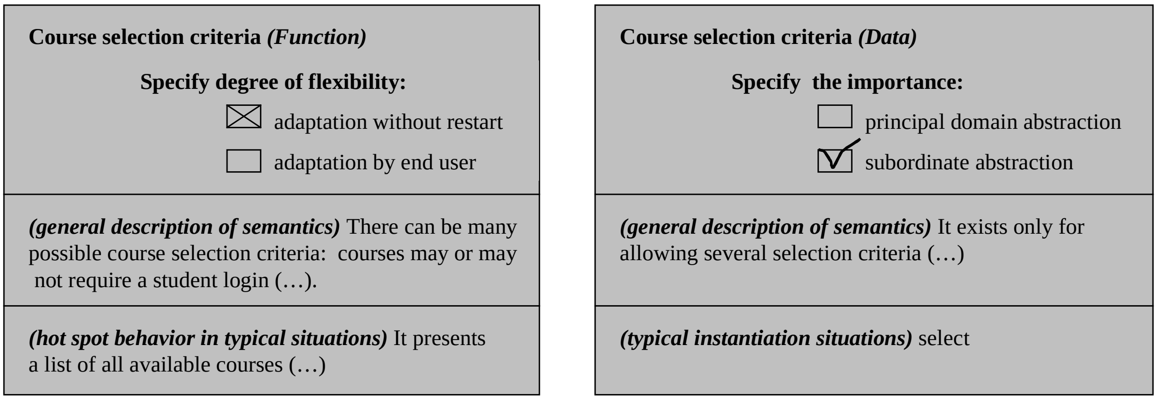


Figure 6. Using hot-spot cards to model the select course hot-spot

Hot-spot cards are primarily concerned with capturing domain knowledge to assist with framework development, and are mainly 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 variation points during design [19, 20]. 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: variation points are not explicitly identified, unless the framework user knows which meta-patterns have been applied.

2.6 Other approaches

The hook tool [11, 12] uses an extended version of UML to model frameworks in which the variation point classes are represented in gray. Although this differentiation between kernel and variation points 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 variation point without any support. If instead of basing itself on standard OO constructs, new elements to model the variation point semantics were provided by the design language, this task could be simplified and partially automated.

Another extension to standard OOADMs that explicitly represents variation points is described in [27]. However, as in the hook approach, the language extensions proposed there are based on existing OO constructs and the variation point semantics are not represented as first-class citizens.

3. Extending UML to model frameworks

This section outlines some requirements for an adequate framework design language. The UML extensions are 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.

3.1 Requirements for a design language

Based on the study of current solutions for framework design and on our experience in modeling and implementing frameworks [1, 8, 10, 24] 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 variation points so as to differentiate them from the framework kernel: The design language should assist the framework user in more easily identifying the parts of the framework that need to be changed when creating applications, during framework instantiation;

R.3. To classify and represent the variation points with respect to their type: This classification should identify the variation point semantics and serve as a basis for the selection of the more appropriate modeling technique for each variation point type, during framework implementation;

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 variation point for calculating final grades in a education framework should only return numbers between 1 and 5;

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 Modeling example

In the ALADIN web-based education framework [10] the class *Actor* 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 7.

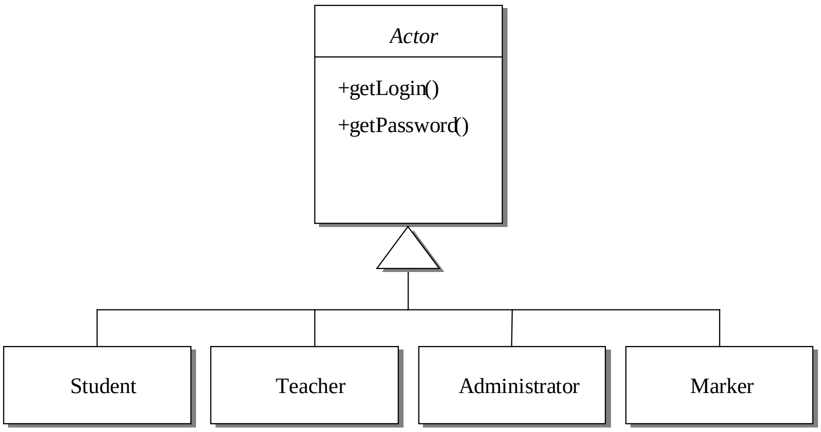


Figure 7. Actor hierarchy

This subsection illustrates how the student subsystem and the actor class hierarchy of the ALADIN framework [10] may be modeled using the UML extensions. Figure 8 is a representation of this part of the framework that represents and classifies the variation points 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 *{variable}* and *{extensible}* extend the UML class definitions to identify two different kinds of variation points: variable methods and extensible classes, respectively. In this example *{variable}* refers to *selectCourse()*, indicating that its implementation may vary 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 class *Actor* as an extensible interface, 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 *NewActorType* is an example of an instance class. Incomplete allows several instance classes to be created from a given extensible interface during the framework instantiation.

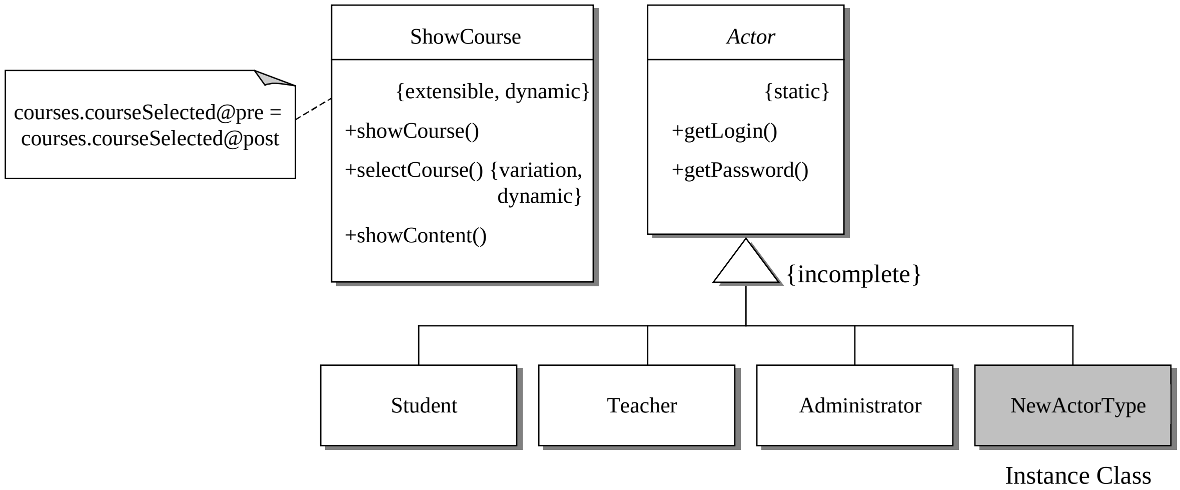


Figure 8. Extended class diagrams

Incomplete is a constraint provided by the standard set of UML constraints, which specifies that not all children of a given 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, applications created from the framework always have three kinds of actors, students, teachers, and administrators, but several actor types may be defined depending on the application specific requirements.

The tagged value *{dynamic}* complements the variation point definition by informing that it may require run-time instantiation. Each variation point must be identified either by the *{dynamic}* or by the *{static}* tagged value, where this last one is used to indicate variation points that do not need run-time instantiation. Variable methods are instantiated by providing the method implementation. Extensible classes are instantiated by the addition of new methods. Extension interfaces are instantiated 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 UML standard class diagrams 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 1 is not required, simplifying the design model.

The design language provides two ways for specifying semantic restrictions on the variation points: notes and template diagrams.

The note attached to the *SelectCourse* extensible class is an OCL [17, 26] 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 the behavior of variation points. An example is presented in Figure 9(a), where the *{optional}* tagged value indicates interactions that may not occur, and depend upon the framework instantiation *(loginPage* and *validateData*). Figure 9(a) tells us that a concrete method that instantiates *selectCourse()* 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 9(b), this step is also a variation point;

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, variation points can be composed of other variation points. Figure 9(b) describes instantiation restriction that applies to *selectionPage():* it cannot use more than HTML 3 frames.

In order to keep template diagrams simple, more than one diagram may be provided for each variation point. In this case, one of them has to be followed by the methods defined during instantiation. If no template diagram or OCL specification is provided the variation point 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 variation points.

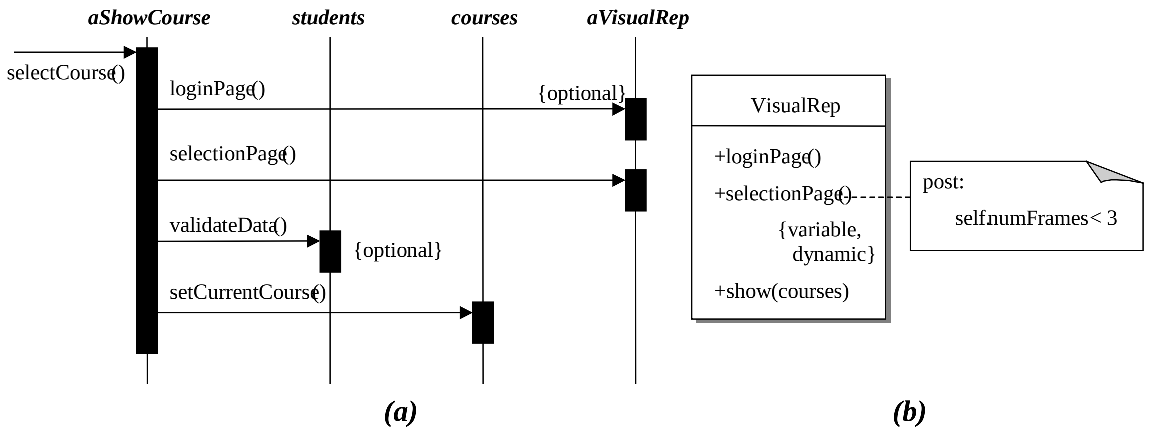


Figure 9. Template diagram (a) and OCL specification (b)

Figure 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 [26]. Each action state is used to represent one variation point. The transitions indicate the order variation points should be instantiated.

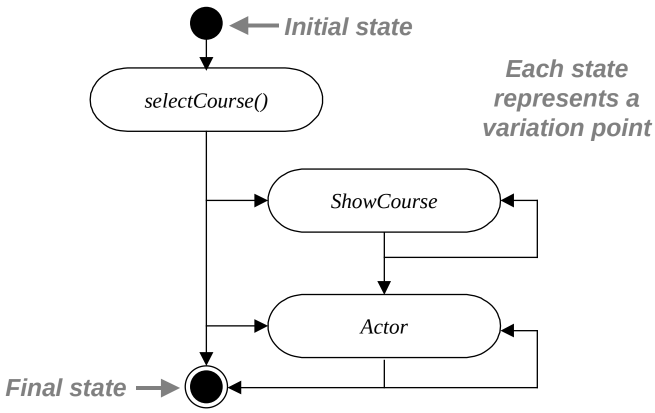


Figure 10. Instantiation diagram

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 *Actor*: several *Actor* subclasses may be defined by a given application but none may be needed by another. The *selectCourse()* variable method 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 variation points, their 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. 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.

3.3 Language description

There are three kinds of variation points in object-oriented frameworks: variable methods, extensible classes, and extensible interfaces. Variable methods are methods that have a well-defined signature, but whose implementation may vary depending on the system instantiation. In the example *selectCourse()* is a variable method. Extensible classes are classes that may have their interfaces extended during the framework instantiation. The *ShowCourse* class, for example, may require the addition of new methods (like *tipOfTheDay())* for each different instantiated application. Extensible interfaces are interfaces or abstract classes that allow the creation of concrete subclasses during the framework instantiation. The instantiation of this last kind of variation point 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 variation points have different semantics: in variable methods the method implementation varies, in extensible classes the class interface varies, finally, in extensible interfaces the types in the system vary (new instance classes may be provided). All three kinds may either be static (do not need to be instantiated at run-time) or dynamic (require run-time instantiation).

These new constructs are represented as extensions to UML by:

- Extending the class diagrams to explicitly identify and classify the variation points;
- Defining template diagrams to 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, *variable, extensible*, *static* and *dynamic.* The first two represent variable methods and extensible classes, respectively. *Static* and *dynamic* are used to classify them regarding to their run-time requirements. The applicability of the *incomplete* constraint has been modified to identify extensible interfaces.

The *restricted* constraint 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 extensible interfaces are extensible classes while subclasses of restricted extension interfaces are standard classes. The use of *restricted* will be exemplified in the case study section.

The keywords *extensible, variable, incomplete,* and *restricted* indicate what are the variation points 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 informally defines their semantics.

Name of extension	Type of ext-ension	Applies to notational element of UML	Informal semantics
Instance Class	Stereotype	Class	Classes that exist only in the framework instance. New instance classes are defined during the framework instantiation.
Variable	Tagged value	Method	It means that the method implementation depends on the framework instantiation. It is used to identify variable method.
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

			extensible classes.
Static	Tagged value	Extensible Interface, Variable Method, and Extensible Class.	It means that the variation point does not require run-time instantiation.
Dynamic	Tagged value	Extensible Interface, Variable Method, and Extensible Class.	It means that the variation point may require run-time instantiation.
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 extensible interfaces.
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 extensible interfaces.

Table 1. Summary of the new elements and their meanings

OCL specifications [17, 26] may be written on notes as in standard UML, however, they have a different semantics if the notes are attached to variation points. In the case of variable methods, it means that all method implementations that may be defined during instantiation should follow the specification. In the case of extensible classes, it applies to all methods that might be added during instantiation. In the case of extensible interfaces, it applies to all methods that may be overridden and added to each instance class.

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 kinds of variation points, just like another kind of OCL specification. For each variation point a finite set of template diagrams may be provided, in which case at least one of them must be followed by the variation point instances. Template diagrams and OCL notes may be used in conjunction for a given variation point, 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 variation point instances, as shown in Figure 9(a). OCL specifications, on the other hand, are generally used to specify invariants that should be satisfied by the variation point instances, as shown in Figure 8 and Figure 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. All variation points should be modeled in the framework instantiation diagram, unless the ones that are instantiated only during run-time.

3.4 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 variation points 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.

The 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 variation point in the framework design. The use of OCL [17, 26] 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 UML extensions are adequate for modeling frameworks. The case studies section emphasizes this point by showing how the language can be applied to model real-world frameworks, assisting the systematization of several tasks in the framework development process.

4. Systematic development

This section shows how the proposed design language can assist framework development by highlighting important aspects of the variation points 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 framework implementation more systematic. It does not attempt to prescribe the best implementation technique to be used with frameworks. A tool that assists this process and other implementation techniques are described in [9].

4.1 Design patterns

Most published design patterns [3, 13, 28] are implementation techniques for variation and extension problems. This subsection shows how the framework design language proposed in this paper can help the selection of the best design patterns to implement each framework variation point. 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 [20] for implementing variable methods and the Visitor design pattern [13] for implementing extensible classes are examples of appropriate pattern implementations for variation points. Examples of design patterns based on abstract coupling include Abstract Factory, Builder, Command, Interpreter, Observer, State, and Strategy [13, 20].

Figure 11 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 extensible class *ShowCourse*. Note that Figure 11 is an *implementation diagram* that uses design patterns to model the variation points of the *design diagram* shown in Figure 8. The use of the Decorator design pattern [13] to implement variable methods is another possible approach, as shown in [9].

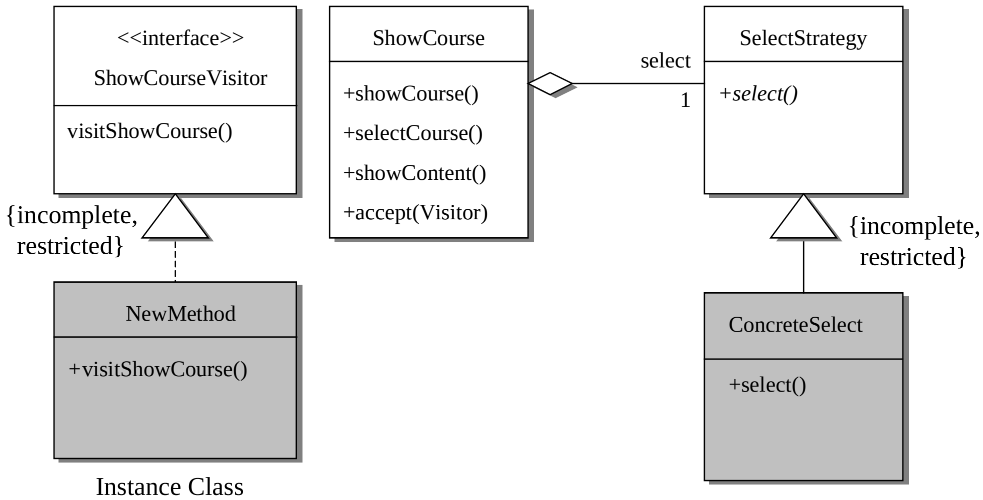


Figure 11. Using design patterns to model framework variation points

Implementing variation points with the selected design patterns does not guarantee that the OCL specifications and the template diagrams (Figure 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 Method design pattern [13], 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 variation point 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 variation points can easily discover how the instantiation process should proceed (which classes to inherit from, which methods to override, and so on).

4.2 Meta-level programming

Meta-level programming [15], as discussed in section 2.2, is a suitable technique for implementing dynamic variable methods. 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 for *selectCourse()* can profit from the template diagram shown in Figure 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 variation point 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.

5. Case studies

This section shows how the framework design language may be applied to assist the development of frameworks in two domains: a framework for the edition of (software) process activities, called PROC [8], and a framework for virtual marketplaces, called V-Market [24]. Several other case studies are presented in [9].

5.1 PROC

PROC uses hierarchical Petri Nets to model the software process activities, 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 12 illustrates a process description, in which each state is represented by a name and the number of tokens it consumes.

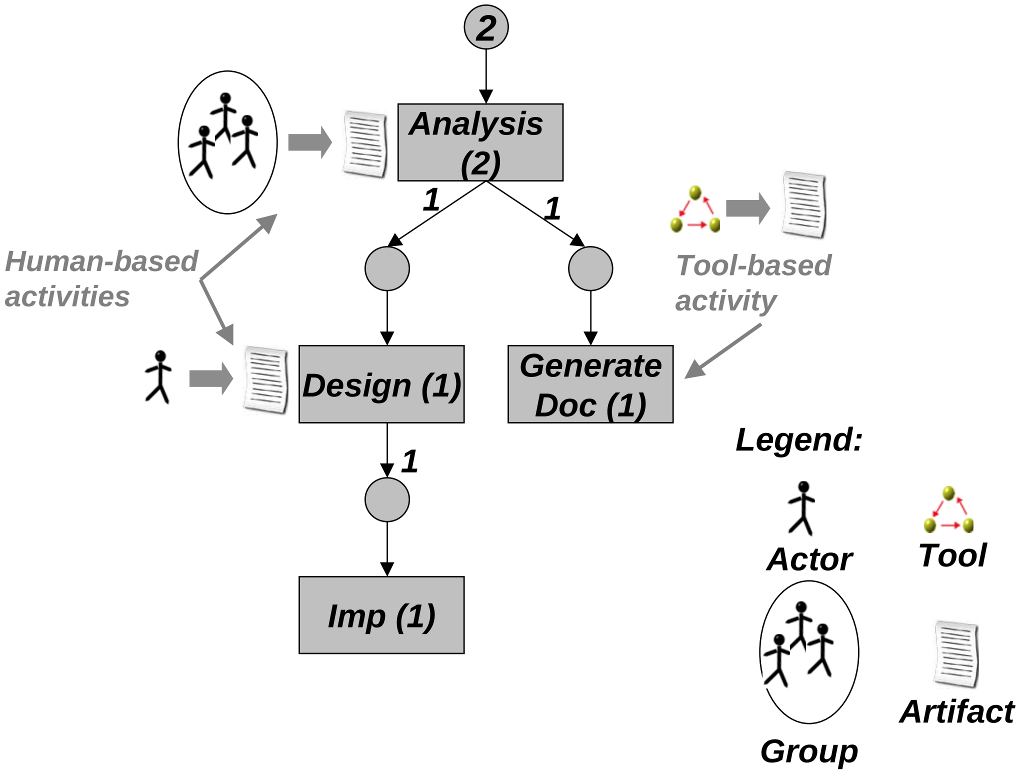


Figure 12. Process description example

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 13 illustrates what happens during the execution of a human-based activity.

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

There are several variation points that must be completed in order to generate an instance of PROC.

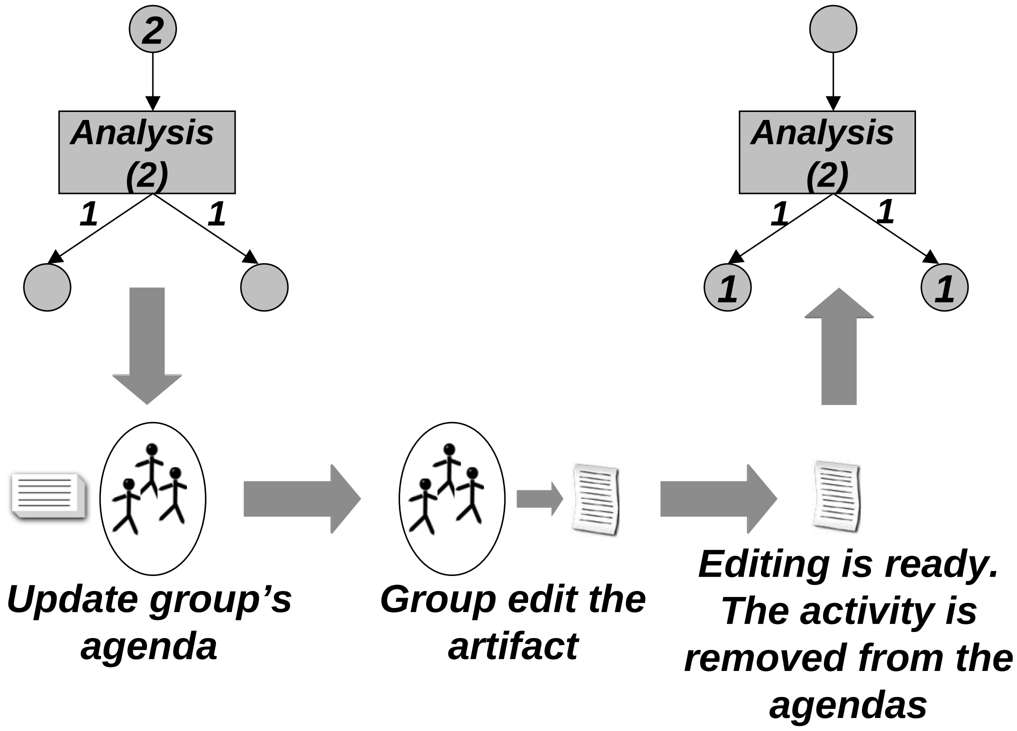


Figure 13. Executing the analysis activity

Group, role, and actor properties: the properties that define an actor or a group may vary from one instantiation of PROC to another. Figure 14 illustrates these variation points;

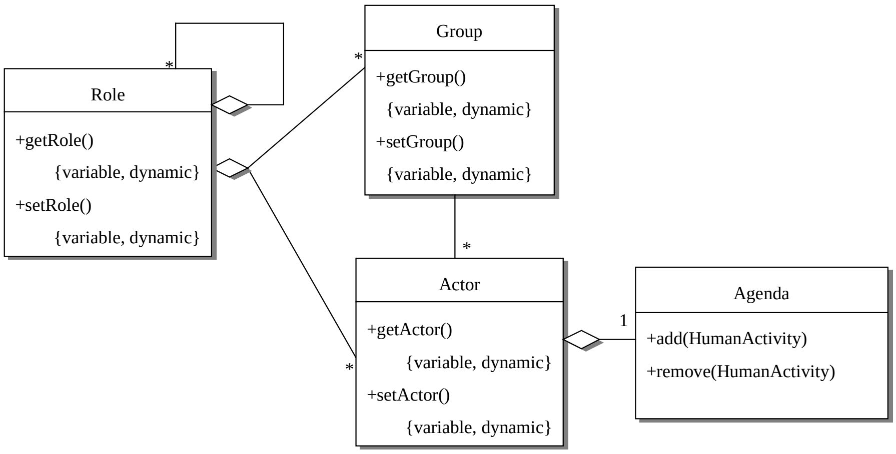


Figure 14. Group, role, and actor variation points

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 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 15 shows how activities and artifacts are defined in the system and how the field type variation point is modeled. Two variable methods are provided by the framework *(equal()* and *greater())* but new methods that operate on *FieldType* may be defined, making it an extensible class;

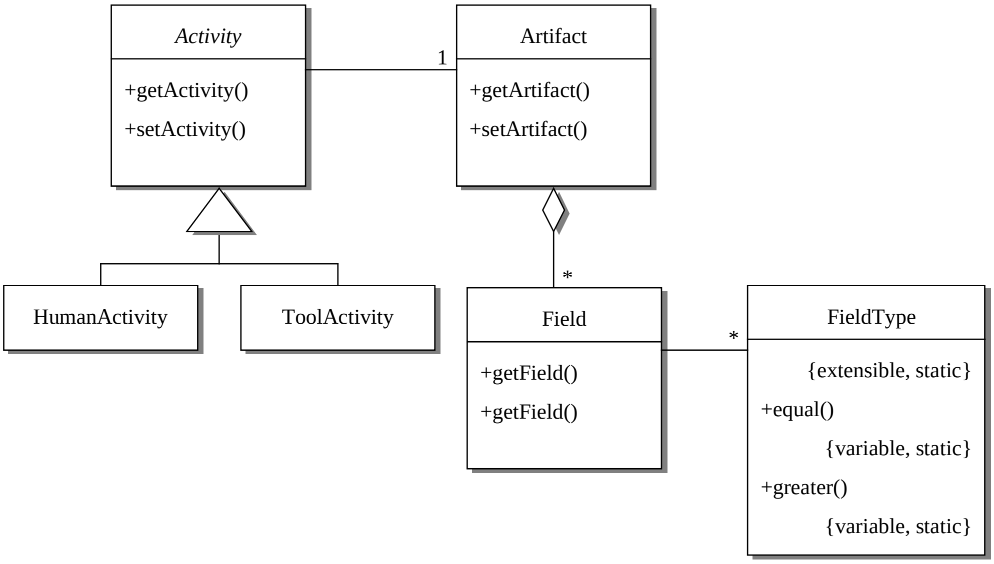


Figure 15. Activities, artifacts, and the field type variation point

Tools: tools are procedures that update the content of artifacts. Each framework instance must define its own tools. For each tool defined in the system the method *fill()* must be defined. The tool variation point is illustrated in Figure 16;

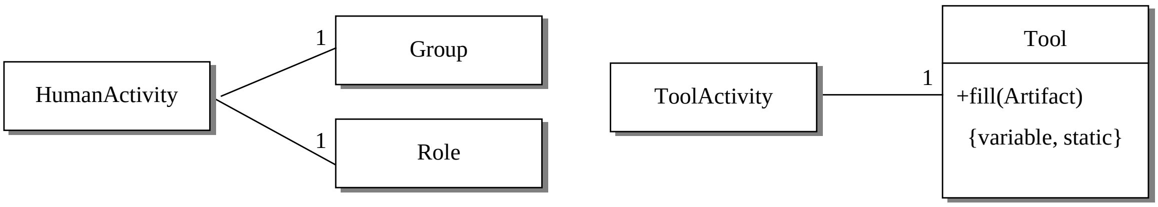


Figure 16. Human activities, tool activities, and the tool variation point

Exception: exceptions are methods that can modify the configuration of the process state, possibly changing the Petri Net's token positions. Each PROC instance must define its own exceptions, as shown in Figure 17;

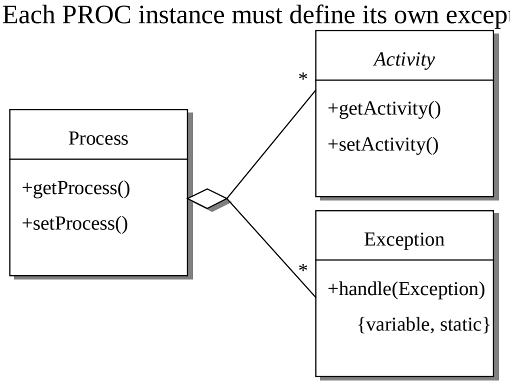


Figure 17. Process and the exception variation point

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 18 presents the system enactor. Each time a process is executed a new *ProcessInstance* object is created. Figure 18 also shows the classes that model the static and dynamic views.

The extended class diagrams presented so far specify the variation points and their semantics. The next step is to provide implementation solutions for each one of them.

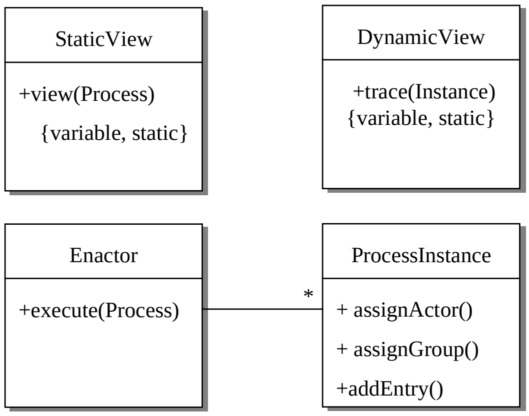


Figure 18. Process enactor, static views and dynamic views

Role, Group, and *Actor* are variable methods that have a limited behavior. The *get* and *set* functions must vary as their properties vary. Also they need to be instantiated dynamically. 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 and automatically applied to the three variation points using the supporting tools described in [9]. The diagram generated is illustrated in Figure 19. The automatic application of MOPs to model these variation points was only possible because they were explicitly marked in the framework design.

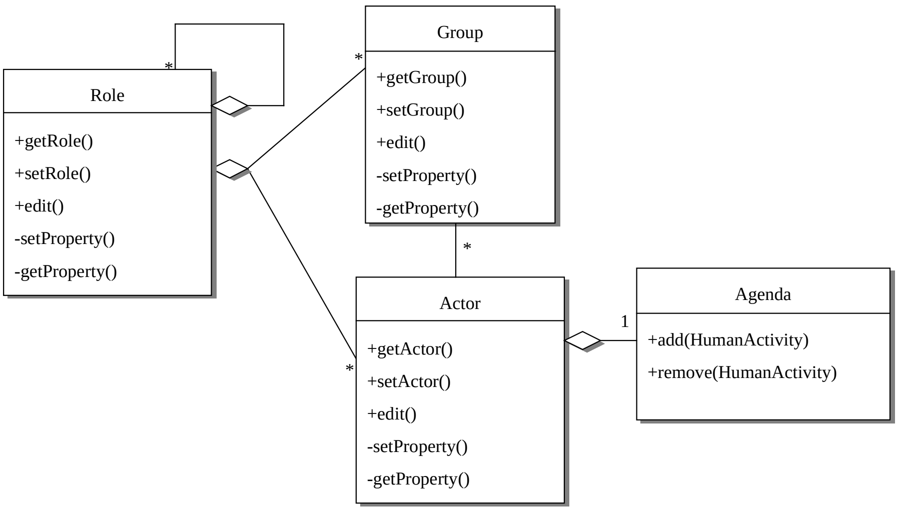


Figure 19. Group, role, and actor implementations

FieldType is an extensible class since for each framework instance several comparison methods can be defined. The *equal()* and *greater()* comparisons must be defined 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 variation point was the use of restricted extensible interfaces combined with the Visitor pattern, as shown in Figure 20. The abstract class *FieldType* provides a default implementation for the methods *equal()* and *greater()*.

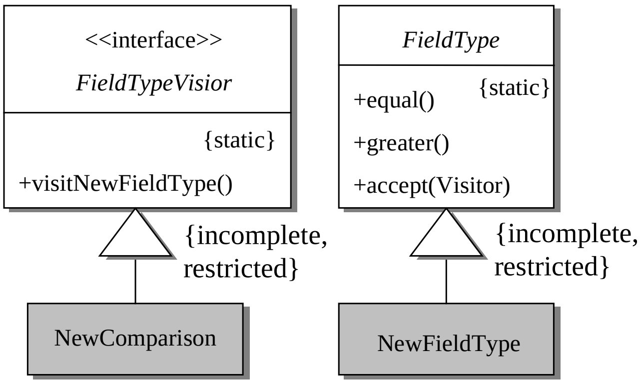


Figure 20. FieldType implementation

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 can be automatically generated from the framework design every time new field types are defined this problem does not occur. An advantage of this solution is that its structure allows new comparisons to be defined in the Visitor hierarchy independently of the existing types. The other four variation points have been modeled using restricted extensible interfaces, as illustrated in Figure 21.

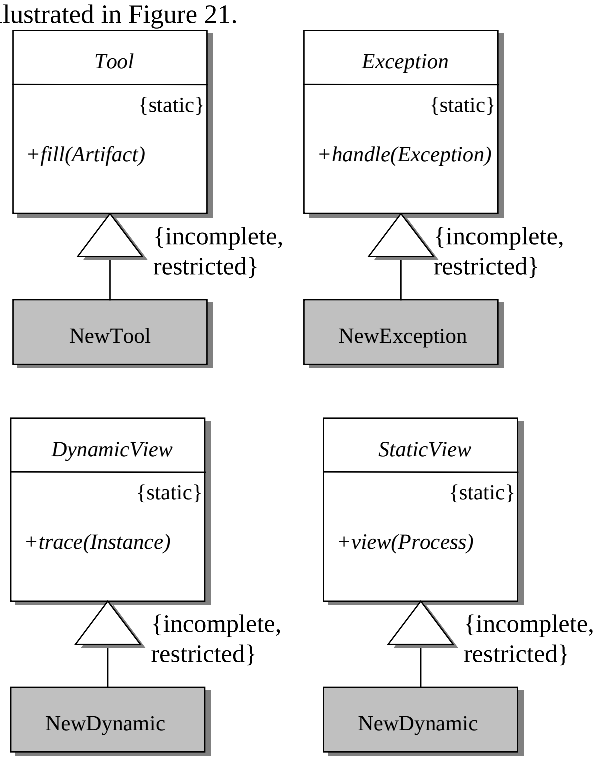


Figure 21. Restricted extensible interface-based implementation

Figure 22 presents the PROC instantiation diagram. Note that at least one *FieldType* must be defined in the system. The other variation points *(Tool, Exception, DynamicView* and *StaticView)* are optional. *Role, Group,* and *Actor* are not modeled in the instantiation diagram since they have been implemented using MOP and their instantiation takes place only during run-time.

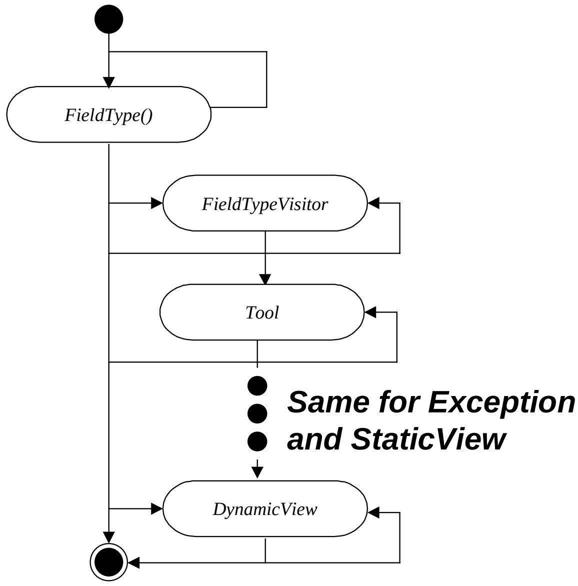


Figure 22. PROC instantiation diagram

The PROC variation points were completely generated from the framework design using the supporting tools described [9], which are able to process the design diagrams and apply code generation transformations. The PROC framework is now completely developed in Java and the development time was reduced by approximately half because of the use of appropriate design techniques and supporting tools.

The documentation of the design using the UML extensions proposed in this paper can help the designer in focusing in what are the variation points. The decision of how they should be implemented is also facilitated (and can be automated by supporting tools), since their semantics are explicitly represented in the design. The instantiation diagrams work as more structure cookbooks [16], which allow the systematization of the instantiation process assisting the framework user.

5.2 V-Market

V-Market is greatly inspired on Media Lab's Kasbah [4] 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 [24]. The most important variation points 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 [4] 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 products to be easily added to the virtual marketplace. For this to be possible, item definitions should be generic enough to support not only commodity type of products such as books and CDs but also intangible ones, such as knowledge about a specific subject, skills, or services. In addition, 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.

To achieve this goal, each item in the framework must be able to store and compare its attributes. Figure 23 illustrates this variation point.

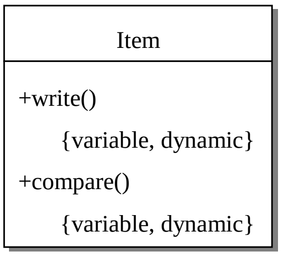


Figure 23. Item variation point

Multiple negotiation dimensions and strategies: Kasbah [4] allows for only one negotiation dimension (price) and three negotiation strategies for this specific dimension: anxious, greedy/frugal, and 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 strategy uses a cubic function (Figure 24).

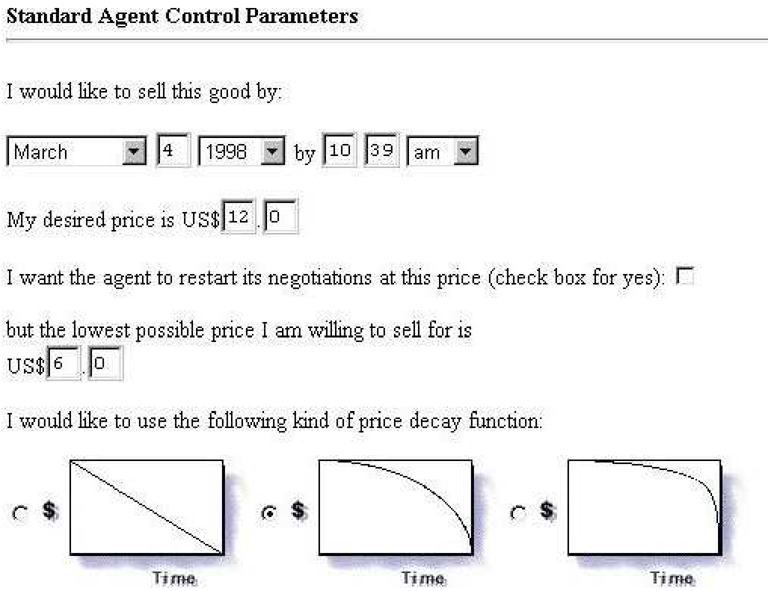


Figure 24. Kasbah negotiation strategies

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 run-time. This type of approach also allows for the creation of negotiation processes for systems that negotiate over more than one dimension[2] as well as to create many different strategies for them.

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

Multiple communication protocols: Kasbah [4] 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 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 25 models these last two variation points, 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.

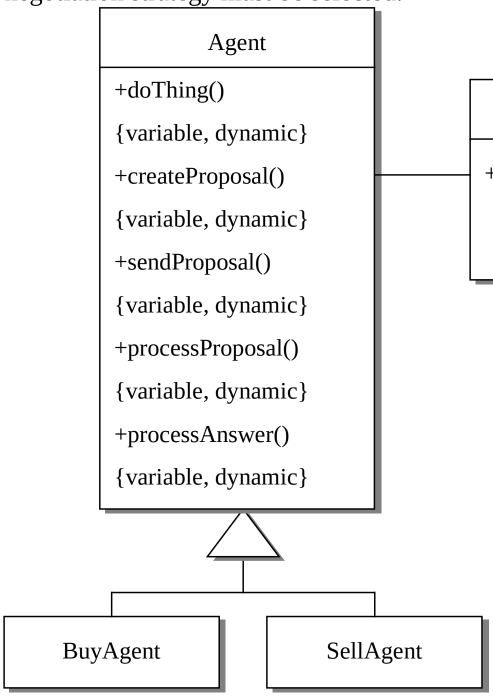


Figure 25. Negotiation strategies and communication protocols

Once the variation points and their semantics are described precisely they need to be implemented using current OO technology.

The item is the most important variation point and should be instantiated easily. The solution adopted was similar to the one used for the *Group* variation point in PROC: an MOP was developed to allow the run-time definition of new items. This solution is shown in Figure 26, in which one *Item* is defined as a list of *MetaItem* objects.

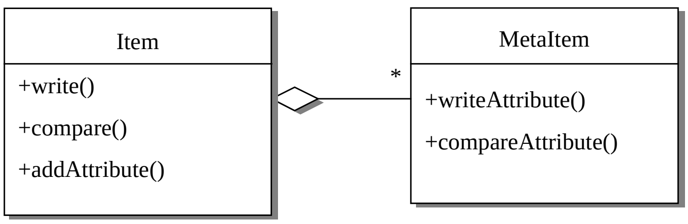


Figure 26. Item MOP

A supporting tool has been defined to enhance the instantiation of this variation point. 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 27 illustrates the DTD used in the instantiation process.

```
<!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 27. Item DTD

The other two variation points have been instantiated through the Strategy and State design patterns as shown in Figure 28. New negotiation strategies and the protocol methods are defined in subclasses of S*trategy* and *State*, respectively.

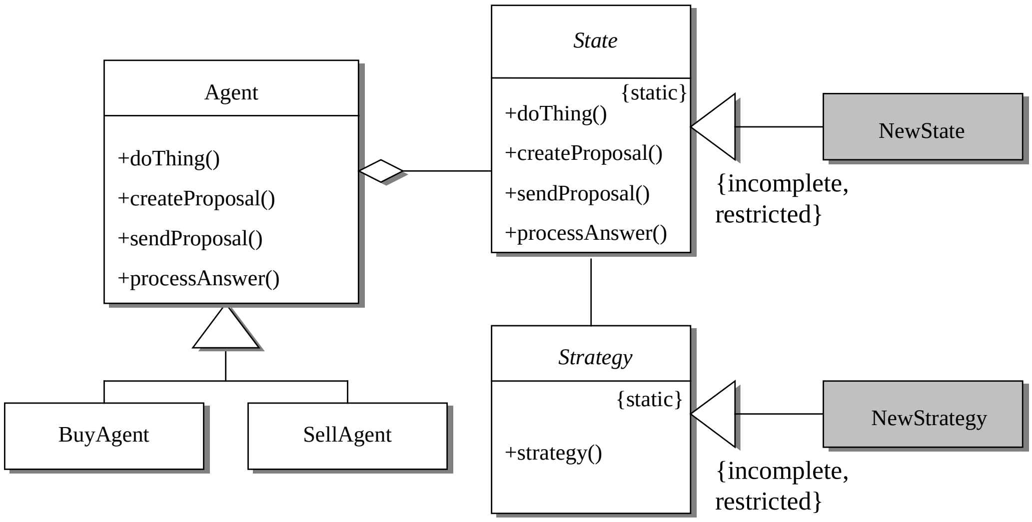


Figure 28. Using Strategy and State to implement variation points

Currently V-Market is completely implemented in Java and a book marketplace instance is being developed to validate the framework. The UML extended design diagrams have greatly improved our ability to instantiate the framework and to develop supporting tools, such as the XML tool for the item variation point.

6. Formal semantics

This section formalizes the UML-extended design language and shows how it can be used to verify framework development and instantiation activities.

6.1 Design language semantics

Figure 29 illustrates the formalization 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 29 shows two design representations: the first, *design*, is a framework since there are several applications that can be instantiated from it; the second, *otherDesign*, is not a framework and *#Meaning(otherDesign) = 1*.

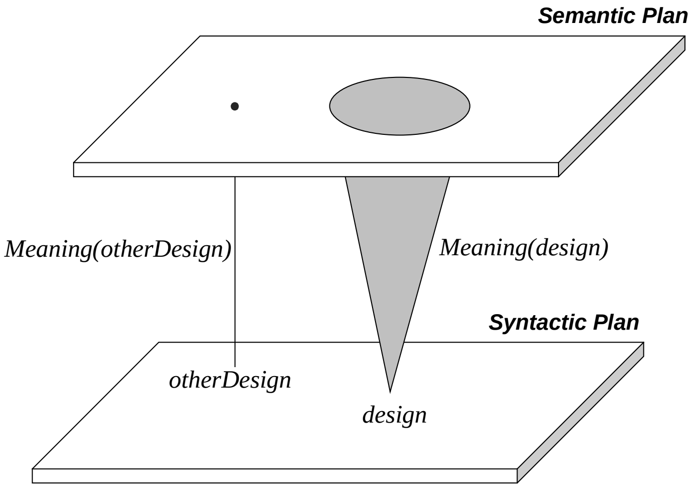


Figure 29. Formalization approach

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 design descriptions of an application and a framework, respectively, then *IsInstance(framework, application)* returns TRUE if *Meaning(application)* $\subset$ *Meaning(framework)* and FALSE otherwise (Figure 30).

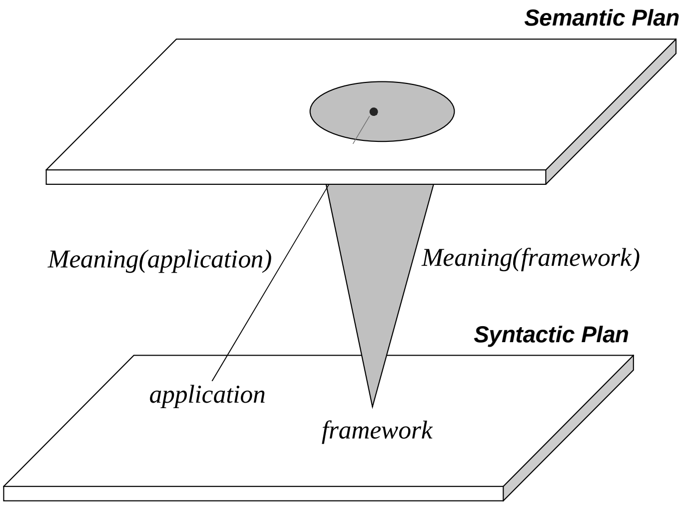


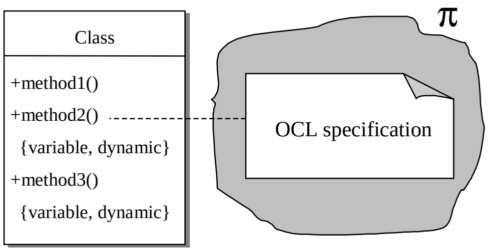
Figure 30. Frameworks and framework instances

Each design described in the framework design language represents a point in the syntactic plan, as shown in Figure 29. 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 variable methods, it is the set of all possible implementations. Variable methods 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 variable methods 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 variation point i. Thus, each implementation h that belongs to H_i satisfies Π_i. Figure 31 illustrates these three possibilities.

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 31 is the set generated by all the combinations of the product $f \times H \times \Phi$, as illustrated in Figure 32.

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



Meaning(method1) = f
Meaning(method2) = H = {h | Π}
Meaning(method3) = Φ

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

Figure 31. Method semantics

Meaning(Class) = $\{ \langle f, h_1, f_1 \rangle, \langle f, h_1, f_2 \rangle, ..., \langle f, h_n, f_k \rangle, ... \}$
where:

Meaning(method1) = f
Meaning(method2) = H = {h | Π}
Meaning(method3) = Φ
$f \in \Phi$
$H \subseteq \Phi$
$\forall i,j \bullet h_i \in H \wedge f_j \in \Phi$

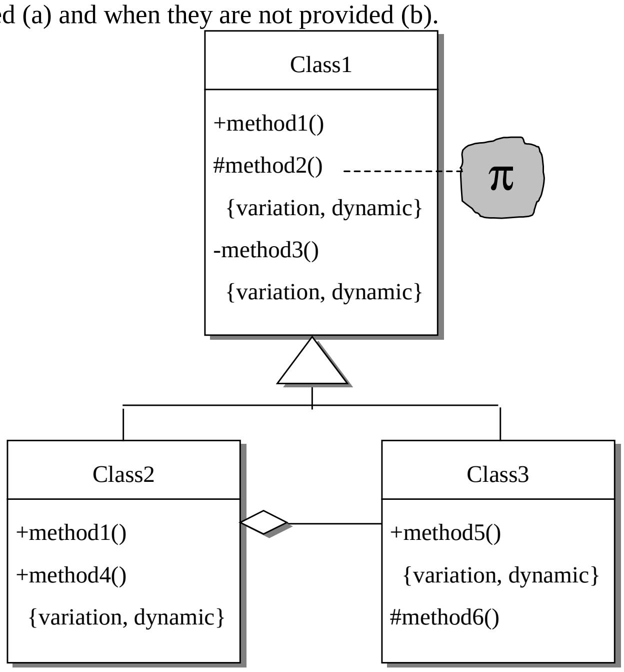
Figure 32. Meaning of an isolated class

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 33. 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. Method overriding is indicated by designating another symbol to represent the method semantics. In this example apostrophes were used for this purpose. UML uses + , #, and – to represent the public, protected, and private access status, respectively.

If none of the methods of a class is a variable method, its symbolic description has the form $f_i \times f_j \times ... \times f_n$. 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 33, the meaning of the design has the symbolic representation *Meaning(Class1) x Meaning(Class2) x Meaning(Class3)*.

Extensible classes contain a possibly infinite number of methods, since its interface may be extended during 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 extensible 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 34 illustrates the semantics of extensible classes when instantiation restrictions are provided (a) and when they are not provided (b).



$$\text{Meaning(Class1)} \approx \text{Class1.}\ f_1 \text{ x Class1.H x Class1.}\Phi$$
$$\text{Meaning(Class2)} \approx \text{Class2.}\ f_1\text{' x Class1.H x Class2.}\Phi$$
$$\text{Meaning(Class3)} \approx \text{Class1.}\ f_1 \text{ x Class1.H x Class3. }\Phi \text{ x Class3. } f_6$$

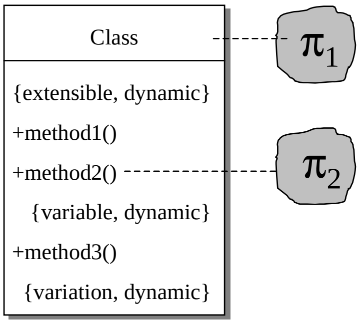
where:

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

Figure 33. 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, extensible interfaces allow the definition of new classes during 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 35 illustrates how the semantics of a design that contains only one extensible interface is calculated, with (a) and without instantiation restrictions (b).

For other methods
that might be added



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

where:

$\text{Meaning(method1)} = f_1$

$\text{Meaning(method2)} = H_2 = \{h \mid \Pi_2\}$

$\text{Meaning(method3)} = \Phi$

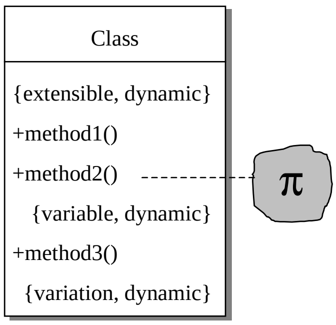
$f_1 \in \Phi$

$H_2 \subseteq \Phi$

$H_1 = \{h \mid \Pi_1\} \wedge H_1 \subseteq \Phi$

(a)

For other methods
that might be added



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

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)

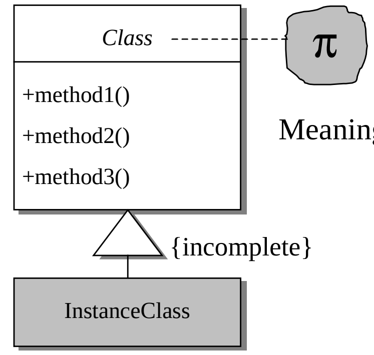
Figure 34. Extensible class semantics

Figure 36 shows the semantics of a design of an instantiated application that was defined from the framework modeled in Figure 35. 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.

Restricted extensible 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 extensible interfaces, as illustrated in Figure 37. It should be clear that the instance classes derived from standard extensible interfaces are extensible classes, while the ones derived form restricted extensible interfaces are not.

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

The extensions *{static}* and *{dynamic}* do not need to be taken in account in the definition of the language semantics since they do not change the set of applications that may be instantiated.



For method1 For method2 For method3 For other methods that might be added

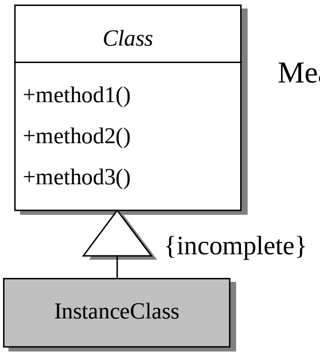
$$\text{Meaning(design)} \approx \{\text{Class. H x Class. H x Class. H x Class. H}^*\}^*$$

Many new classes might be added

where:

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

(a)



For method1 For method2 For method3 For other methods that might be added

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

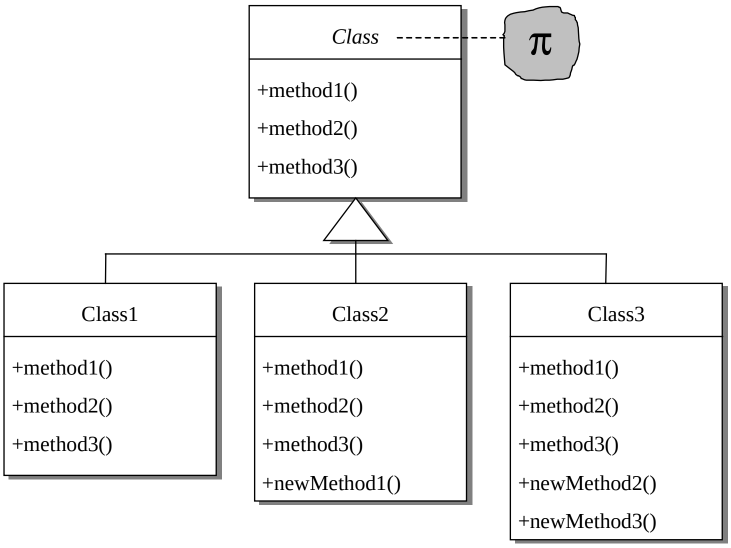
Many new classes might be added

where:

$$\text{Meaning(method1)} = \Phi$$
$$\text{Meaning(method2)} = \Phi$$
$$\text{Meaning(method3)} = \Phi$$

(b)

Figure 35. Meaning of extensible interfaces



$$\begin{aligned}\text{Meaning(instance)} \approx\ & \{\text{Class.}f_1\text{' x Class.}f_2\text{' x Class. }f_3\text{' }\}\text{ x}\\ & \{\text{Class.}f_1\text{'' x Class.}f_2\text{ x Class. }f_3\text{ x Class. }h_1\ \}\text{ x}\\ & \{\text{Class.}f_1\text{ x Class.}f_2\text{'' x Class. }f_3\text{'' x Class. }h_2\text{ x Class.}h_3\}\end{aligned}$$

where:

$$H = \{h \mid \Pi\} \wedge H \subseteq \Phi$$
$$\forall i,j \bullet h_i \in H \wedge f_j \in \Phi.$$

Figure 36. Instantiation example

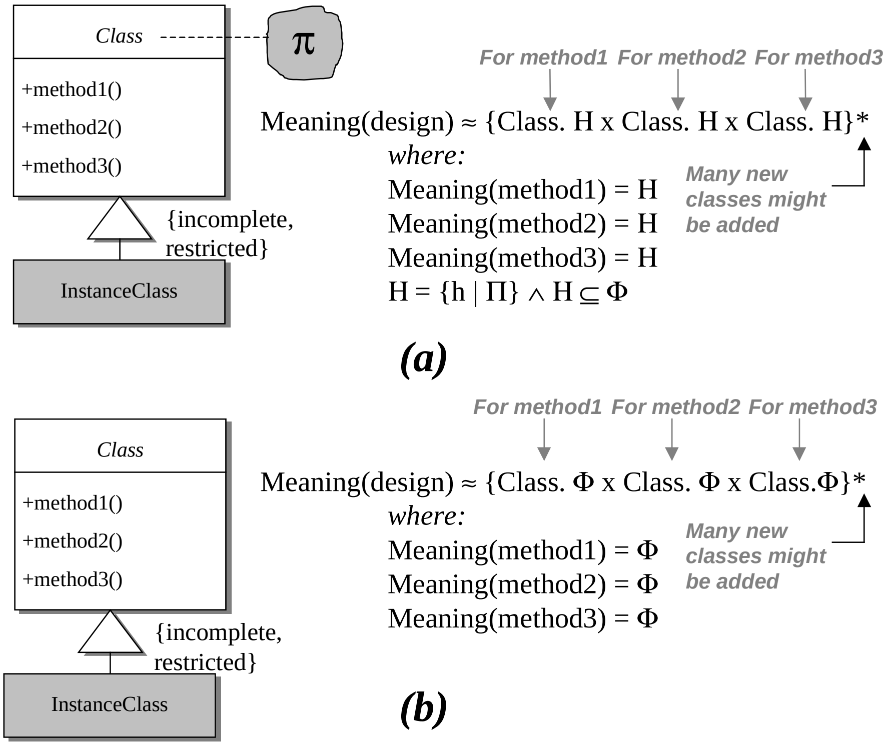


Figure 37. Semantics for restricted extensible interfaces

6.2 Verifying framework development

Figure 38 illustrates part of the web-based education framework [10] and the symbolic representation of its design[3]. This section will use this framework as an example to illustrate how the use of design techniques to model variation points may be formally verified. The mappings between design and implementation diagrams will be defined as transformations. These transformations have to be behavior-preserving, since the implementation approach chosen for the variation point must not change the framework semantics.

Figure 39 illustrates a framework generated from Figure 38 design representation, using the implementation approach based on the Strategy and Visitor patterns [13] to model variable methods and extensible classes, respectively. It also presents the symbolic representation of the semantics of the generated framework.

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

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

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

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

In this example there are two transformations that must be proved: (i) variable method-Strategy and (ii) extensible class-Visitor. Thus, the above expression may be split into the following expressions:

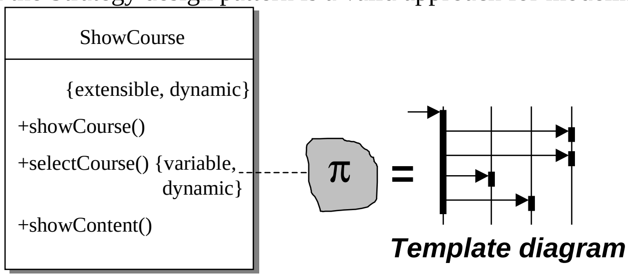
$$\text{(i) } \{SC.\ H\} = \{SC.f_3\} \times \{SS.H\}^* \text{ and (ii) } \{SC.\Phi^*\} = \{SC.\ f_4\} \times \{V.\Phi\}^*$$

In (i) the method *selectCourse()* uses delegation to invoke method *select()*, in one of the *SelectStrategy* subclasses. Thus, the implementation $\{SC.f_3\}$ just passes the control to one of the $\{SS.H\}^*$ instances. Knowing that, expression $\{SC.f_3\} \times \{SS.H\}^*$ may be simplified to $\{SC.H\}^*$. Finally, as just one *SelectStrategy* subclass is

---

[3] This is a variation of the design presented in Figure 8 that does not consider the *Actor* hierarchy and instantiation restrictions for the *ShowCourse* class.

invoked at a given time the repetition symbol may be eliminated. Thus, we can conclude that this transformation is sound and the Strategy design pattern is a valid approach for modeling variable methods.



Meaning(design) ≈ {SC.f_1 x SC.H x SC.f_2 x SC.Φ*}

where:

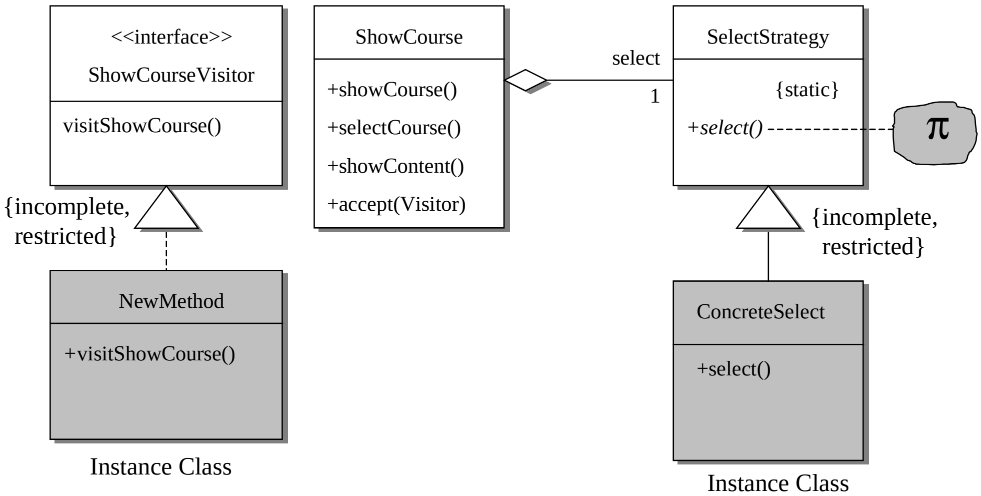
Meaning(ShowCourse.showCourse) = SC. f_1

Meaning(ShowCourse.selectCourse) = SC. H

Meaning(ShowCourse.showContent) = SC. f_2

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

Figure 38. The web-based education framework and its semantics



Meaning(design) ≈ {SC.f_1 x SC.f_3 x SC.f_2 x SC. f_4 } x {SS. H}* x {V. Φ}*

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. Φ

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

Figure 39. Semantics of design pattern-based implementation

A geometric view of this proof is presented in Figure 40. 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 variation point semantics.

In (ii) 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 $\{SC.\ f_4\}\ x\ \{V.\Phi\}^*$ can then be simplified to $\{SC.\Phi\}^*$, which is equivalent to $\{SC.\Phi^*\}$. This means transformation (ii) is a behavior-preserving transformation, and the Visitor design pattern may be used to model extensible classes.

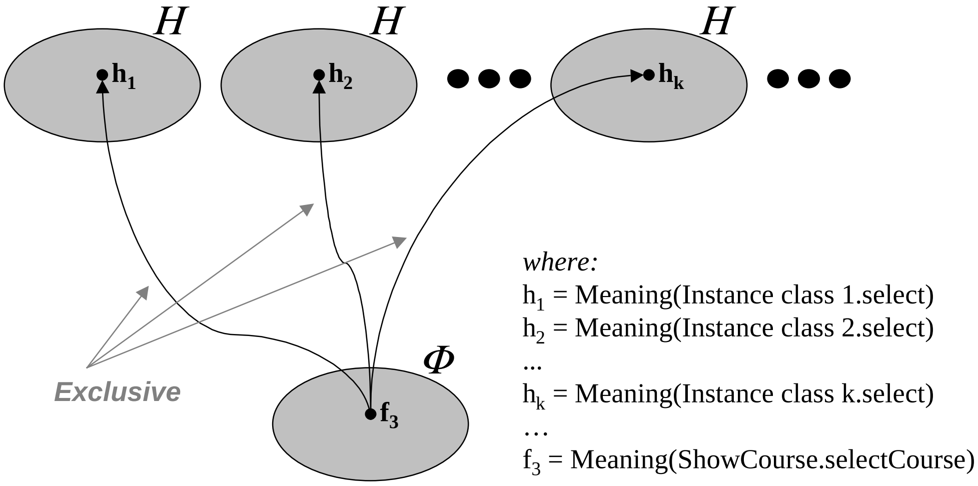


Figure 40. Geometric view of the variable method-Strategy transformation

A geometric view of this proof is presented in Figure 40. 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 variation point semantics.

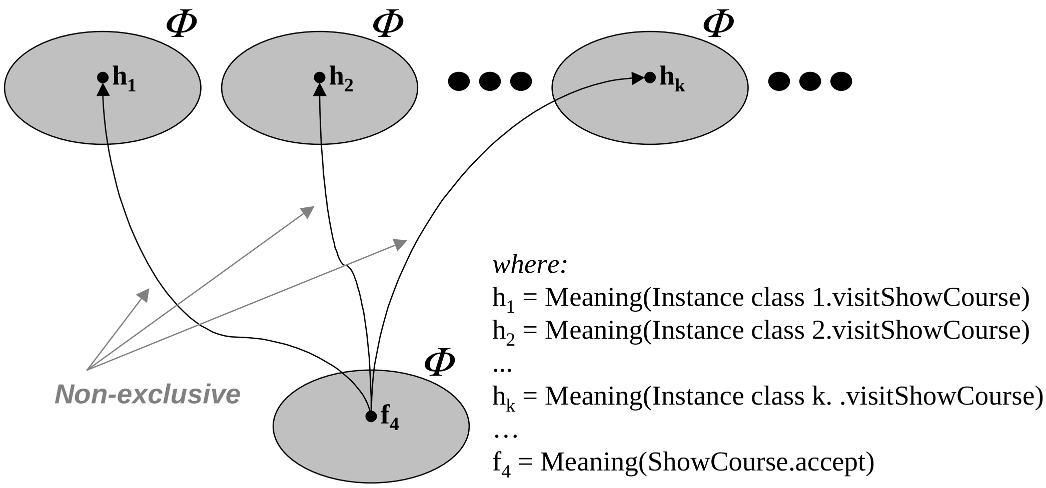


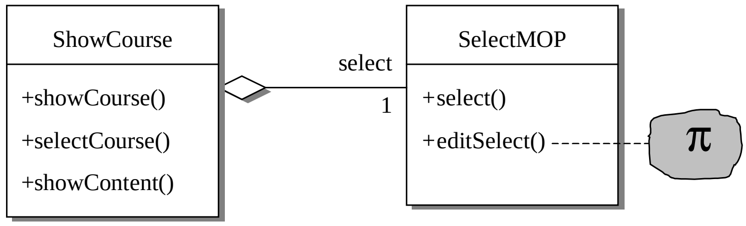
Figure 41. Geometric view of the extensible class-Visitor transformation

As discussed in section 4, meta-object protocols [15] may be used to implement variable methods that have instantiation restrictions. Figure 42 shows how the variable method, *selectCourse()*, may be implemented by an MOP and presents the symbolic representation of the design semantics.

The variable 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 $\{MOP.f_5 \times MOP.f_6\}$ expression may be reduced to $\{MOP.H\}$, which is equivalent to $\{SC.H\}$, concluding the proof. Figure 43 presents a geometric view of this mechanism.



$$Meaning(design) \approx \{SC.f_1 \times SC.f_3 \times SC.f_2\} \times \{MOP.f_5 \times MOP.f_6\}$$

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 42. Semantics of meta-object protocols

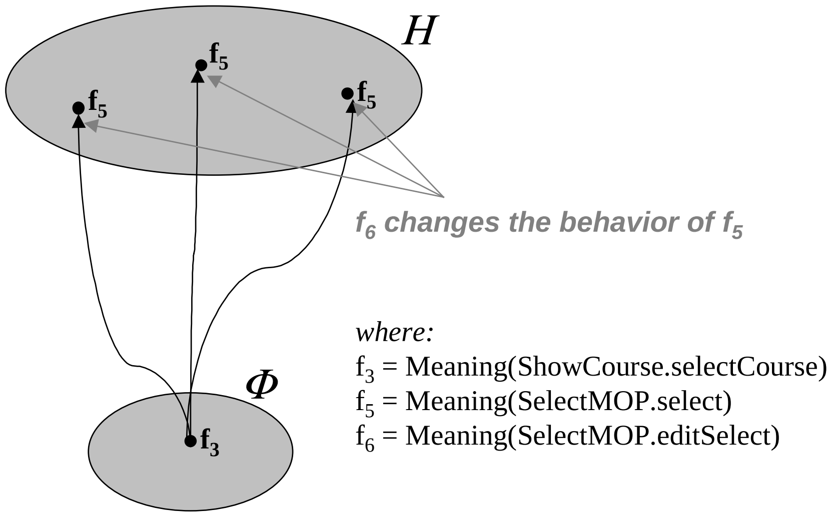
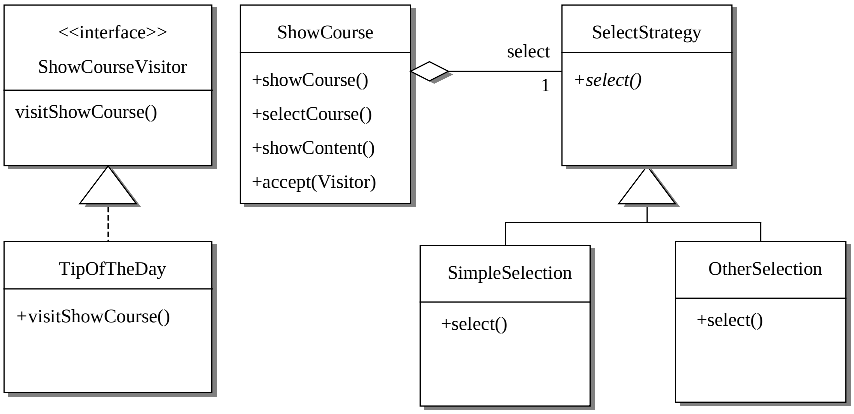


Figure 43. Geometric view of the MOP mechanism to implement variable methods

6.3 Verifying framework instantiation

Framework instantiation is the process of transforming the framework into concrete applications. These transformations are not behavior-preserving, since *Meaning(application)* ⊂ *Meaning(framework)*. However, they may be checked through the *IsInstance* function. Figure 44 illustrates one possible instance of the Figure 39 framework and the symbolic representation of their semantics.



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

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 44. 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:

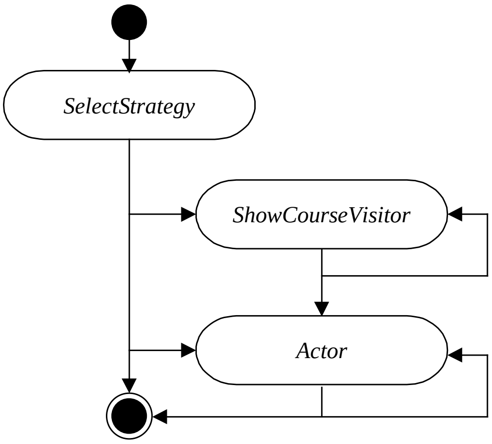
$$\{SS.f_5\} \text{ x } \{OS.f_6\} \text{ x } \{TD.f_7\} \subset \{SS.H\}^* \text{ x } \{V.\Phi\}^*$$

Knowing what instance classes have been added to what variation points, the equation may be split into two simpler ones:

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

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, supporting tools can list the set of equations that must be true for the design to be considered a framework instance, as shown in [9].

However, if the instantiation diagram presented in Figure 45 is the one that defines the instantiation process of the framework presented in Figure 39, the application shown in Figure 44 would not be a valid framework instance. The reason is that this instantiation diagram limits *SelectStrategy* to only one subclass.



$\#\{SC.H\}^* = 1$
$\exists\, x \mid x \in \{V.\Phi\} \rightarrow \exists\, y \mid y \in \{A.\Phi\}$

Figure 45. 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 45. 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.

The Verifier tool, described in [9], calculates the framework semantics and assists the verification of development and instantiation transformations.

7. Conclusions and future work

Several approaches have been used to model frameworks, including standard OOADMs [25, 26], meta-patterns [19, 20], and role diagrams [22]. However, none of these approaches explicitly represent and classify framework variation points. 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 variation points. Instantiation diagrams are proposed to represent the instantiation process.

Although the proposed extensions have been used to model several frameworks successfully, they represent an initial version of the language and the modification of the existing extensions and creation of new ones is a topic for future investigation. The development of new case studies and the collaboration with other research groups will help us to refine the current UML extensions and develop new ones. The use of connectors [7] and generic parameterized classes are now being investigated.

Several other case studies that further illustrate the language applicability are presented in [9], together with a description of a set of supporting tools.

One of the most important claims of this paper is that frameworks should first be modeled through an appropriate design language and the selection of design patterns, or other possible techniques, to model the variation points 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. We are now developing an integrated tool that adequately supports development activities by representing frameworks through the UML extended constructs. We believe that the UML extensions and the supporting tools can be very effective in enhancing framework development. They build on the sound basis of UML standardization and propose no radically new path.

The formalization of the language semantics shows 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.

8. References

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

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

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

4. 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.

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

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

7. S. Ducasse and Tamar Richner, “Executable Connectors: Towards Reusable Design Elements”, ESEC’97, *LNCS 1301*, Springer-Verlag, 483-500, 1997.

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

9. M. Fontoura, “A Systematic Approach for Framework Development“, Ph.D. Thesis, Computer Science Department, PUC-Rio, 1999 (http://www.les.inf.puc-rio.br/~mafe).

10. 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).

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

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

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

14. D. Hamu and M. Fayad, ”Achieving Bottom-Line Improvements with Enterprise Frameworks“, *Communications of ACM*, 41(8), 110-113, 1998.

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

16. 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.

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

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

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

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

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

22. D. Riehle and T. Gross, “Role Model Based Framework Design and Integration”, OOPSLA’98, *ACM Press*, 117-133, 1998.

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

24. 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.

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

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

27. 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).

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

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