A Framework Design and Instantiation Method Based on Viewpoints

Marcus Felipe M. C. da Fontoura, Edward H. Heausler, and Carlos José P. de Lucena
Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 225, 22453-900 Rio de Janeiro, Brazil
e-mail: [mafe, hermann, lucena]@inf.puc-rio.br

PUC-RioInf.MCC35/98 Outubro, 1998

ABSTRACT

This paper describes a viewpoint-based design and instantiation method for object-oriented frameworks. Case studies have shown that high levels of software reuse can be achieved through the use of object-oriented frameworks. Although, analysis and design methods such as design patterns, meta-object protocol, refactoring, class reorganization, and behavior of modification for framework have been proposed to support framework development there are still problems related to these approaches. The method proposed in this paper uses the concept of viewpoints and the hot-spot relationship in object-oriented design to guide the designer on the identification of hot-spots in the structure of the framework and in the generation of the design patterns that implement each hot-spot. The paper also shows how the use of domain specific languages (DSLs) can help in the framework instantiation process. The domain specific language captures domain concepts and helps the framework user to create an application in an easier way, without concerning for implementation decisions while remaining focused on the problem domain. The specification written in the DSL is then translated to the framework instantiation code through the use of transformational systems. A case study in the Web-based Education domain is presented to illustrate the method utilization. A formal specification of the design method using the Z specification language is also presented. Through this formalization it is possible to precisely describe each method's step and to highlight its most important properties.

KEY WORDS: object-oriented frameworks, framework design, framework instantiation, viewpoints, hot-spot relationship, design pattern essentials, hot-spot cards, domain specific languages, Web-based Education domain, Z specification language.

RESUMO

Este artigo descreve um método para design e instanciação de frameworks baseado em viewpoints. Pesquisas anteriores têm mostrado que altos níveis de reutilização de software podem ser alcançados através do uso de frameworks. Apesar de métodos como design patterns, meta objetos, refactoring, reorganizção de classes e modificação de comportamento terem sido propostos para auxiliar o desenvolvimento de frameworks, ainda existem problemas associados a esses métodos. O método proposto neste artigo usa o conceito de viewpoints e a relação de hot-spot para ajudar o designer na identificação dos hot-spots na estrutura do framework e na geração de design patterns para cada um dos pontos de flexibilização. Este artigo também descreve uma técnica que captura os conceitos do domínio em linguagens de domínio (DSL), ajudando o usuário do framework a criar aplicações específicas de maneira mais simples, sem se preocupar com decisões de implementação e permanecendo focado no domínio do problema. A especificação escrita na DSL é então transformada no código de instanciação do framework através de sistemas transformacionais. Um estudo de caso em Educação baseada na Web é apresentado para ilustrar a utilização do método. Uma descrição formal do método de design usando a linguagem Z também é apresentada. Através dessa formalização é possível descrever precisamente cada um dos passos do método e provar suas propriedades mais importantes.
PALAVRAS CHAVES: frameworks orientados a objetos, design, instanciação, viewpoints, relação hot-spot, meta-patterns, hot-spot cards, linguagem específica de domínio, ambientes para educação baseados na Web, Z.

1. INTRODUCTION

Prior research has shown that high levels of software reuse can be achieved through the use of object-oriented frameworks [15]. An object-oriented framework captures the common aspects of a family of applications. It also captures the design experience required while producing applications, and thus, it allows designers and implementers to reuse their experience at the design and code levels.

Although object-oriented software development has experienced the benefits of using frameworks, a thorough understanding of how to identify, design, implement, and change them to meet evolving requirements is still object of research [36]. Techniques such as design patterns [10, 23], meta-object protocols [17], refactoring [16], and class reorganization [4] have been proposed to support framework development and cope with some of the challenges.

This paper presents a design method for object-oriented software that combines frameworks and viewpoints. The method uses viewpoints and the hot-spot relationship as key design concepts for building the framework. The hot-spot relationship supports the integration of frameworks and design patterns, a new and challenging issue [29].

The framework instantiation process is also presented. The basic idea behind it is to capture the domain concepts in a domain specific language (DSL) [12], which will help the framework user build the instantiation code in an easier without concerning for implementation decisions while remaining focused on the domain specific problem. The specification written in the DSL is then translated to the framework instantiation code through the use of transformational systems [19, 5].

A case study in the Web-based Education domain is presented to illustrate the method utilization.

The method formalization is achieved through the specification of each artifact and process involved. The specifications, presented as Z schemas [32, 33], are used to precisely describe each of the method's steps and to highlight its most important properties.

2. FRAMEWORK DESIGN METHOD

A framework [15, 24] is defined as generic software for an application domain. It provides a reusable semi-finished software architecture that allows both single building blocks, and the design of sub-systems to be reused.

The approach to framework design presented in this paper is based on the idea that any framework design can be divided into two parts: the kernel sub-system design and the hot-spot sub-system design. The kernel sub-system design is common to all the applications that the framework may generate, and the hot-spot sub-system design describes the different characteristics of each application that can be supported by the framework. The hot-spot sub-system uses the method and the information provided by the kernel sub-system and may extend it.

Figure 1 illustrates the viewpoint-based design method through the identification of its artifacts and processes. The following sub-sections describe each of the elements involved.

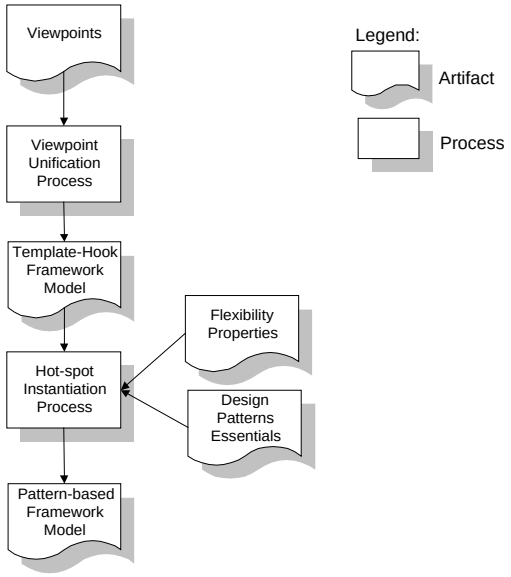

Figure 1. Viewpoint-based development method process

2.1 VIEWPOINTS

The development of complex software systems involves many agents with different perspectives of the system they are trying to describe. These perspectives, or viewpoints [9], are usually partial or incomplete. Software engineers have recognized the need to identify and use this multiplicity of viewpoints when creating software systems [1, 14].

The first input artifact in the framework design method is a set of design diagrams that are developed based on perspectives, where a perspective defines a possible different use of the framework. In this way, a different system design associated with each different perspective is produced (Figure 2). In this paper the term viewpoint will be used to represent a standard object-oriented design associated with a framework perspective. The design is represented through the use of object-oriented diagrams, which can be developed using any OOADM (object-oriented analysis and methods) notation. This paper uses OMT [27] to illustrate the examples presented.

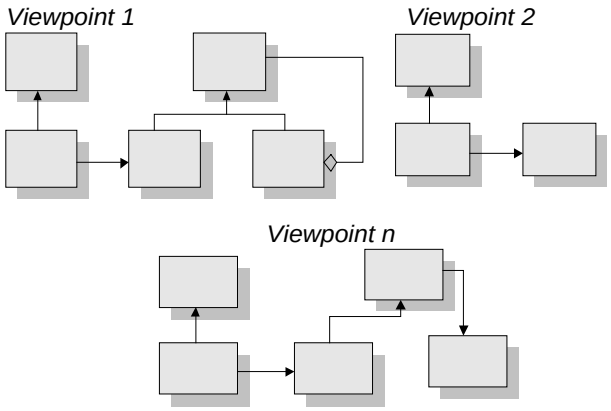

Figure 2. Different viewpoints of the same framework

In [25] Roberts and Johnson state that "Developing reusable frameworks cannot occur by simply sitting down and thinking about the problem domain. No one has the insight to come up with the proper abstractions." They propose the development of concrete examples in order to understand the domain. Our strategy is quite similar, analyzing each one of the viewpoints as a concrete example and deriving the final framework from this analysis.

2.2 THE VIEWPOINT UNIFICATION PROCESS

Once all the relevant viewpoints are defined the kernel design structure can be found by analyzing the viewpoints design representations and obtaining a resulting design representation that reflects a structure common to all viewpoints. This common structure is the "*unification*" of the viewpoints. This part of the design method is based on the domain-dependent semantic analysis of the viewpoints design diagrams to discover the common features of the classes and relationships present in the various viewpoints. The common part will compose the kernel design sub-system.

The elements that are not in the kernel are the ones that vary, and depend on the use of the framework. These elements define the framework hot-spots [23, 28] that must be adaptable to each generated application. Each hot-spot represents an aspect of the framework that may have a different implementation for each framework instantiation.

The unification process can not automatically generate the final framework design since the viewpoints represent concrete applications, and for this reason the semantics of how to define the flexible part of the final framework is not present in their design. The design patterns essentials proposed by Pree [24] provide the appropriate semantics to the framework flexibility, where the software engineer can select the best design pattern constructor for each of the framework's hot-spots, considering their flexibility requirements.

A new relationship in object-oriented design, called hot-spot relationship, is used in the unification process to represent the relationships between kernel and hot-spot objects. The semantics of this new relationship is given by the design patterns essentials. This implies that the hot-spot relationship is in fact a meta-relationship that is implemented by a design pattern that is generated taking into account the hot-spot flexibility requirements.

A more detailed description of the viewpoint unification process is now presented. This process imposes some pre-conditions on the viewpoints that are going to be unified:

- The viewpoints need to have name consistency, which means that classes and methods with different names represent different concepts, and for this reason are not unified;
- The viewpoints' structure must be consistent. In this way, two classes with same name and signature can not be related by different kinds of relationships in different viewpoints, as shown in Figure 3.

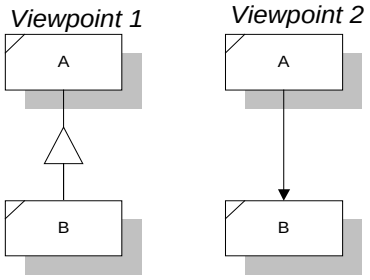

Figure 3. Viewpoint inconsistency

In some cases class restructuring approaches [4, 16] can handle the inconsistency. When the set of viewpoints is consistent they can be unified. The result of the unification process is the template-hook framework model[1]. The unification process is based on the following rules:

1. Every class that belongs to the set of viewpoints has a corresponding class, with same name, in the template-hook framework model;
2. If a method has the same signature and implementation in all the viewpoints it appear it has a corresponding method, with same name, signature, and implementation[2], in the template-hook framework model;

[1] Which is based on template and hook classes [24].

[2] This check cannot be automatically performed, but some approximation techniques can help.

3. If a method exists in more then one viewpoint with different signature it has a corresponding hook method in the template-hook framework model, with same name but undefined signature and implementation;
4. If a method exists in more then one viewpoint with different implementation it has a corresponding hook method in the template-hook framework model, with same name and signature but undefined implementation;
5. All the methods that uses hook methods are defined as template methods. There is always a hot-spot relationship between the class that has the template method and the class that has its correspondent hook method;
6. All the existing relationships in the set of viewpoints that do not have a corresponding hot-spot relationship are maintained in the in the template-hook framework model.

Figure 4 shows an example of two consistent viewpoints that are to be unified.

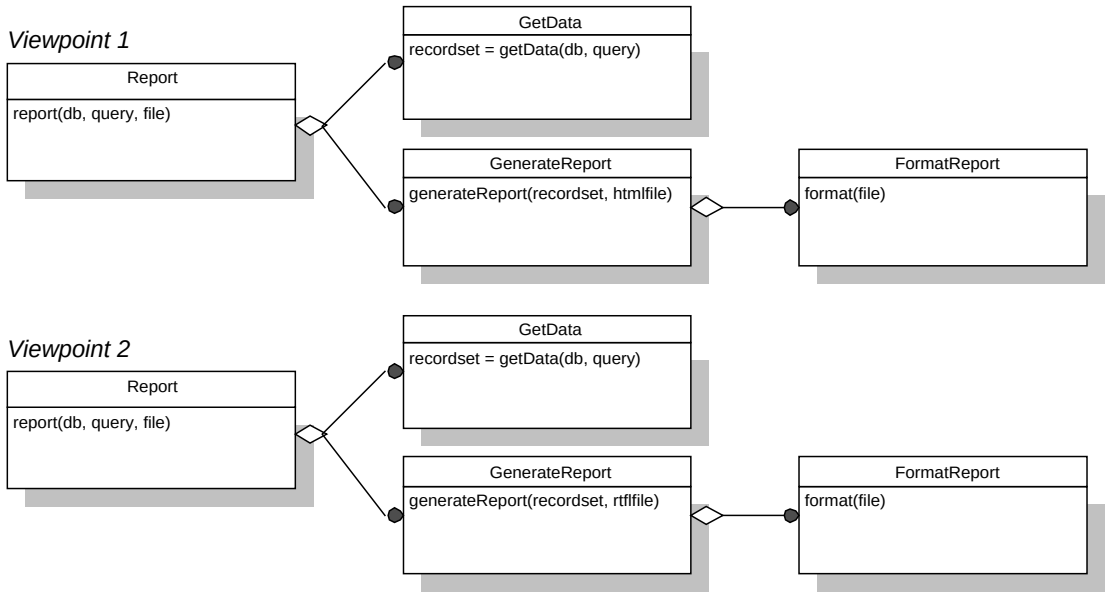

Figure 4. Viewpoint unification example

Viewpoint 1 shows a design diagram of a simple report generator. Its main class is Report, which uses the classes GetData to access the information required by the report and GenerateReport, to generate the final report in an HTML file. The class GenerateReport uses the FormatReport class to configure the layout of the generated HTML file.

Viewpoint 2 follows basically the same structure. The classes Report and GetData have the same methods, with same signature and implementation. The method generateReport (in class GenerateReport) has different signatures in the two viewpoints, since one asks for an HTML file while the other asks for a RTF file. In this case rule 3 implies that it is a hook method. The method format (in class FormatReport) has different implementations in the two viewpoints, since one configures HTML files while the other configures RTF files. Rule 4 defines this method as another hook method in the template-hook framework model. Rule 5 says that all the methods that use hook methods are marked as template methods. So, in this example, the template methods are report and generateReport.

The design presented in Figure 5 is the result of the unification of viewpoints, defined as the template-hook framework model. The dashed arrow represents the hot-spot relationship. Note that the signature of method generateReport is UNDEFINED, as established by rule 3, and the implementations of methods generateReport and format are also UNDEFINED (rules 3 and 4), although not shown in the diagram.

The undefined signatures and implementations are defined latter on, in the method instantiation process (Section 3).

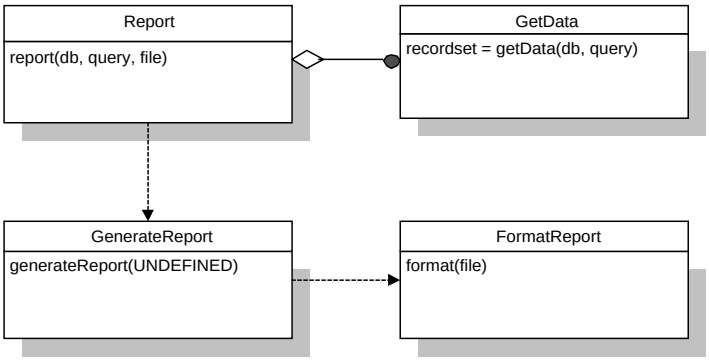

Figure 5. Template-hook framework model

2.3 THE TEMPLATE-HOOK FRAMEWORK MODEL

The template-hook framework model design semantics is given by the set of analyzed viewpoints, or in other words, it copes with all the requirements specified in the original viewpoints. It uses the hot-spot relationship to provide the flexibility semantics, not specified in the original set of viewpoints.

The hot-spot relationship is a new relationship in object-oriented design that combines template and hook classes through the use of the design patterns essentials. We are now working in the formalization of the semantics of this new relationship using category theory [11] and object calculus [8]. Through this formalization we expect to proof the correction of the flexibility properties showing that they can be implemented through the design patterns essentials.

In the template-hook framework model the hook methods can be defined as components [15] that may be plugged into the hot-spots.

2.4 THE HOT-SPOT INSTANTIATION PROCESS

This process uses the artifacts template-hook framework model, flexibility properties, and design patterns essentials as inputs and generates the pattern-based framework model as output. It is responsible for eliminating all the hot-spot relationships from the template-hook framework model, replacing then by the appropriate design pattern. The hot-spot cards guide the generation of an appropriate design pattern that will implement each hot-spot relationship, providing a systematic way for generating design patterns based on flexibility properties.

Figure 6 shows the hot-spot card layout, which is a variation to the one presented in [24]. The two flexibility properties used in the method are shown in the card: adaptation without restart and recursive combination.

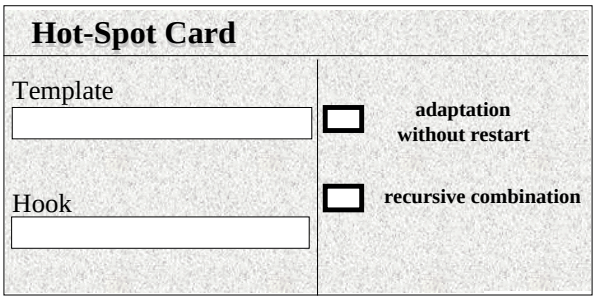

Figure 6. Hot-spot card

The following table shows the mappings between the hot-spot card flexibility properties and the appropriate meta-pattern (or design pattern essential). In the unification meta-patterns the template and hook methods belong to the same class and adaptations can be made only through inheritance, which requires the application restart for the changes take effect. In the separation meta-patterns the template and hook methods appear in different classes and adaptations can be made at run time through the use of object composition.

The recursive combinations of template and hook methods allow the creation of direct object graphs, like the composite design pattern present in the GoF catalog [10].

	Adaptation without restart	Recursive Combination	Meta-pattern
1			Unification
2	☑		Separation
3		☑	Recursive Unification
4	☑	☑	Recursive Separation

Table 1. Mappings between flexibility properties and meta-patterns

As an example, let us consider the hot-spot relationships in the template-hook framework model shown in Figure 5. Suppose that it is necessary that new generateReport methods be defined in the system at run time. Also suppose that the format method do not need to be redefined at run-time. Since neither of these relationships requires a recursive combination the hot-spot cards that represent their flexibility properties are presented in Figure 7.

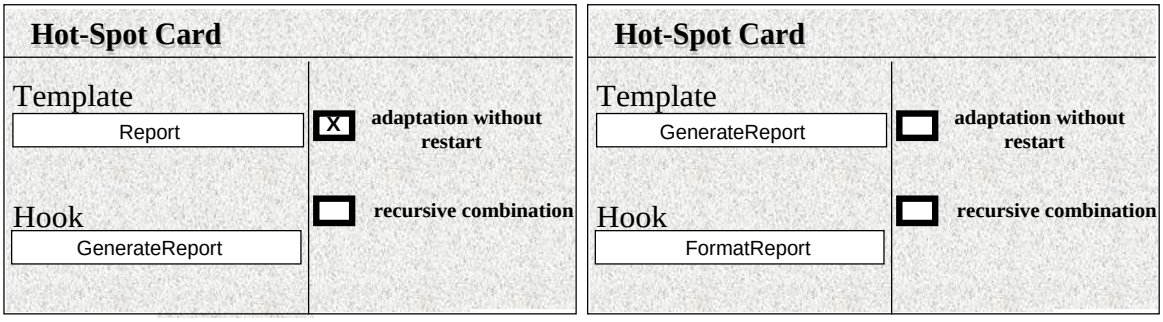

Figure 7. Hot-spot card utilization example

The result diagram after the generation of the appropriate design pattern for each hot-spot relationship is presented in Figure 8, which is the pattern-based framework model.

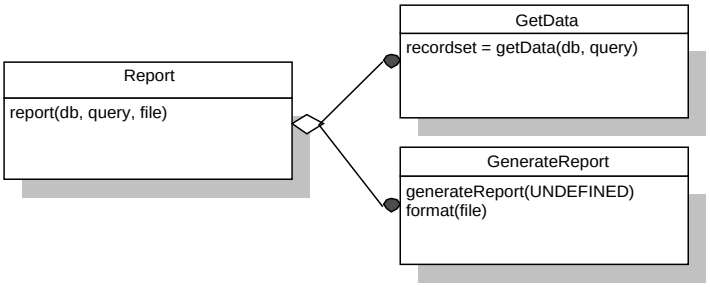

Figure 8. Pattern-based framework model

Since the design pattern generated for the hot-spot relationship between the classes GenerateReport and FormatReport unifies the template and hook methods in the same class and the FormatReport class only have the format method, this class is not needed in the pattern-based framework model. A very important property of the design method is the control of the design complexity, leading to a simple and readable design. Note that, in this example, the generated pattern-based framework model has less classes than each of the original viewpoints.

2.5 THE PATTERN-BASED FRAMEWORK MODEL

The hot-spot instantiation process generates as a result a class diagram based on design patterns, which provide all the flexibility required for the framework's hot-spots. These patterns are the best implementation of each hot-spot relationship present in the template-hook framework model. Section 3 describes the framework instantiation process, showing how the pattern-based framework model is used in the generation of a specific application.

3. FRAMEWORK INSTANTIATION METHOD

The most common way to instantiate a framework is to inherit from some 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 which code and where it should be written since frameworks class hierarchies can be very complex, especially for non-expert users.

Therefore, the "common" instantiation process is not always the ideal way to describe a particular application. This happens because of several facts:

- the language used to build and use the framework is a wide spectrum language, where it is not always easy to express the user intentions [31]. A short example would be the definition of event handlers in Microsoft MFC. The user has to write a macro to register her methods that will be called when a button is pressed or a list is scrolled. With an interface definition language [20], this task becomes easier and more intuitive;
- the complexity of framework class hierarchies and the difficulty of finding the points where the code should be written, that are the framework hot-spots or flexible points.

Our method proposes a different process for framework instantiation, where a particular application is described by domain specific languages (DSLs) [3, 7, 12, 22] that are designed precisely for the framework domain. The use of the DSLs allows the designer to develop quickly and effectively a complete software system [12]. In this way, the technique basic idea is to capture domain concepts in a DSL, which will help the framework user to build code in an easier way, without worrying about implementation decisions and remaining focused on the problem domain. The specification written in the DSL is translated to the framework instantiation code through the use of transformational systems [19, 5].

3.1 DOMAIN SPECIFIC LANGUAGES

A domain specific language is used to formally specify software designs [12]. It is a formal language that is expressive over the abstractions of an application domain. Common examples of DSLs are:

- Tex and Latex, text formatting languages;
- YACC, a parser generator;
- Mathematica, an extensible language for mathematical modeling;
- Schema description and query languages for databases.

An advantage of using DSLs is that the domain expert can express domain specific concepts directly, rather than encoding them. This allows the domain expert to formalize the specification of a software solution immediately instead of communicating the specification informally to a software specialist who may be less familiar with the intended application domain. When designing DSLs two important criteria should be kept in mind:

- The DSL must be intelligible to a domain expert;
- The semantics must allow a specification expressed in a DSL to be translated into effective procedures that realize the specification.

Typically, such specification will provide static and dynamic constrains on an artifact of the application domain, or will specify its dynamic behavior. Often, a syntax driven editor can be used to help an application designer to formulate an application in a DSL.

3.2 THE FRAMEWORK INSTANTIATION PROCESS

This section shows how DSLs can be used to instantiate frameworks in a straightforward manner, making possible the use of complex frameworks by regular developers. To achieve this goal the Draco-PUC [19] transformational system is used to generate framework instantiation code from programs written in the DSLs.

The proposed instantiation process uses the following elements: DSLs, the pattern-based framework model, and a transformational system. The DSLs and the pattern-based framework model gather the domain main concepts. The transformational system is used to map the specification written in the DSLs to framework instantiation code. A customized application is the combination of the pattern-based framework model with its instantiation code. Figure 9 presents a flow diagram of this process.

As a precondition, this process requires the DSLs definition. The Draco-PUC allows the definition of new languages by the specification of its grammar, in a BNF like notation, and a set of transformation rules to generate the framework instantiation code. The domain designer performs these steps. To generate a specific application a regular user needs to write a specification in DSL and then apply the set of transformation rules over it.

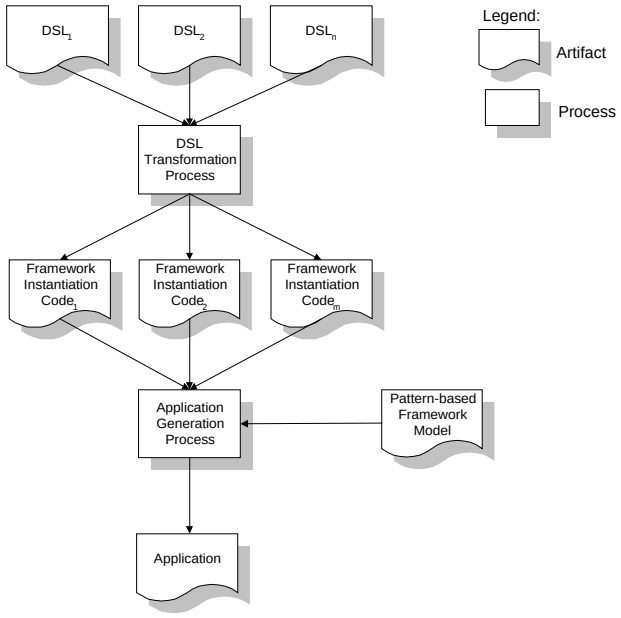

Figure 9. Framework instantiation process

Let us consider again the report generation example, where instantiation code needs to be generated for the generateReport and format methods. Analyzing the initial set of viewpoints we see that variation in the generateReport signature is due to the variation of the report files types that can be generated, HTML and RTF. The variation in the implementation of the format method is also generated by the necessity of configuring the layout of various kinds of report files. Thus, the DSL used in the instantiation process has to have operators to deal with these file type variations. As an example consider the following program written in a DSL for the report generation domain.

```
ReportType := HTML

GenerateReportSpec := {ColMaxLength = 8; ColsPerRow = 10}

FormatLayoutSpec := {UseFrames; Font = ”Times”; BackgroundColor = ”White” }
```

Through the transformation of this code all the signatures and implementations in the pattern-based framework model that are UNDEFINED can be specified. The DSL operator ReportType is used to complete the generateReport method signature, while the code generated by the transformation of the other two operators are used in the implementation of the methods generateReport and format. Figure 10 shows the design structure of the generated application.

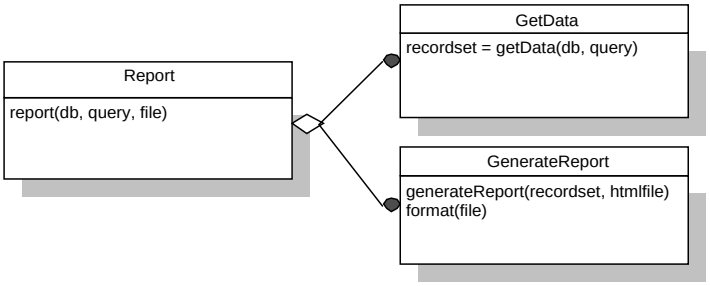

Figure 10. Instantiated application

The functionality provided by the DSL constructors vary according to the selected meta-pattern. Table 2 summarizes this dependency.

Meta-pattern	DSL Semantics
Unification	Must provide the appropriated semantics to generate the code for the hook method.
Separation	Must provide the appropriated semantics to generate the code for the hook method and to specify how the hook method varies during run-time. This DSL type must generate code that will create concrete subclasses from the pattern-based framework model hook class to hold each of the method variations, similar to the Strategy design pattern described in the GoF catalog [10].
Recursive Unification	Similar to the Unification meta-pattern but must also allow the recursive composition.
Recursive Separation	Similar to the Separation meta-pattern but must also allow the recursive composition.

Table 2. DSLs and meta-patterns

Note that in some domains, like geometry and civil engineering, a visual [30] domain oriented language can produce better results than a textual language.

4. A CASE STUDY IN WEB-BASED EDUCATION

This section presents an example of deriving a framework for the Web-based Education domain. The following subsections present the models of the analyzed WBE environments. Each of these models was considered as a different viewpoint of the final object-oriented WBE framework, and the viewpoint unification process was used to define the framework kernel structure.

Two aspects must be considered when analyzing the following models. First, except for AulaNet [21] and LiveBOOKs, we do not know the exact object-model of the analyzed environments. The models presented here were specified by the use of these environments. Second, when the models are similar we just refer to the figure that describes it in order to avoid presenting similar models twice.

Since the objective of our comparison of WBE environments is the definition of a framework, we were not interested in a feature by feature analysis of the environments. We only need to analyze of the core entities of the WBE domain. A useful concept adopted throughout all the elicitation process was the concept of services. We define a service as functionality provided by the environment. Examples of services are discussions groups, course news, quizzes, and bulletin boards.

4.1 AULANET

AulaNet [21] (http://www.les.inf.puc-rio.br/aulanet) is a WBE environment developed in the Software Engineering Laboratory, at PUC-Rio. AulaNet allows several institutions to use the same environment simultaneously. Each institution may have several departments. The courses are related to institutions and each course has assigned actors. A course consists of a selection of services. This design structure is presented in Figure 11.

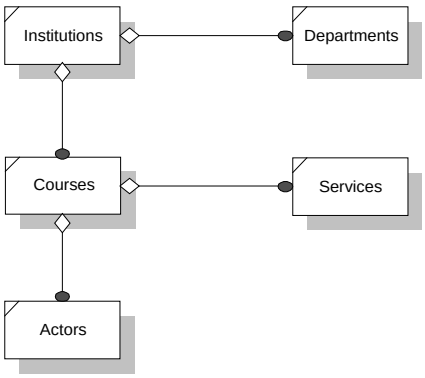

Figure 11. AulaNet OMT class diagram

The AulaNet environment is composed of two sites: a learning site and an authoring site. The students use the learning site to attend a specific course, while the authors use the authoring site to create and maintain the courses. The class structure that implements both sites is shown in Figure 12, where the class idioms represent the support to multiple languages (English, Portuguese, etc).

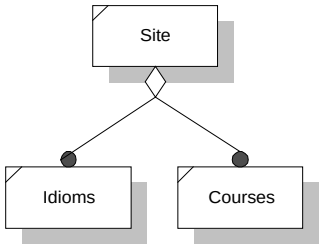

Figure 12. AulaNet site class structure

4.2 LIVEBOOKS

The LiveBOOKs distributed learning/authoring environment is a computer-based teaching/learning and authoring system that has supports learning and authoring activities. LiveBOOK class structure is very similar to AulaNet's, except of two main differences: it is does not support the definition of many institutions and departments and the actors types are not restricted to student and author, as show in Figure 13.

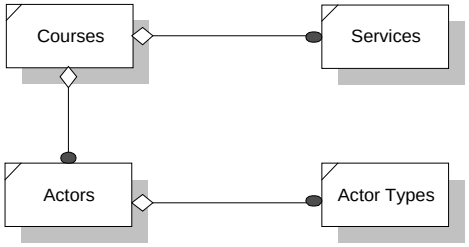

Figure 13. LiveBOOKs class diagram

4.3 WEB COURSE IN A BOX

Web Course in a Box (WCB) (http://views.vuc.edu/wcb/into/wcbintro.html) is a course creation and management tool for Web-based or Web-assisted delivery of instruction. The main difference of this environment and the other two previously presented is that in WCB the final user can modify the visual representation (interface) for each one of its entities as shown in Figures 14 and 15.

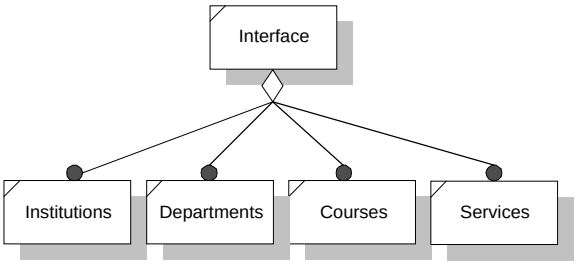

Figure 14. WCB interface class structure

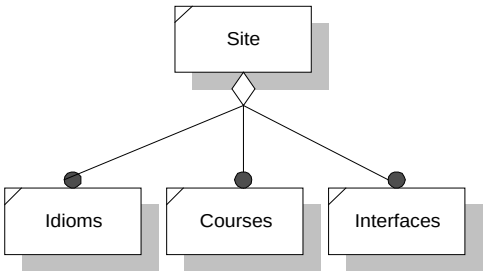

Figure 15. WCB site class structure

4.4 WEBCT

WebCT (http://homebrew.cs.ubc.ca/webct/) is a tool that facilitates the creation of sophisticated WBE environments. The Web-CT class structure can be seen as an extension of the LiveBOOKs. The new concept is

that each service can be of two different types: internal, which is implemented by the environment, and external, which is a WWW application not implemented by the environment (elsewhere in the Internet). Examples of external services are chat applications, CU-SeeMe, e-mail, and list servers. This design structure is presented in Figure 16.

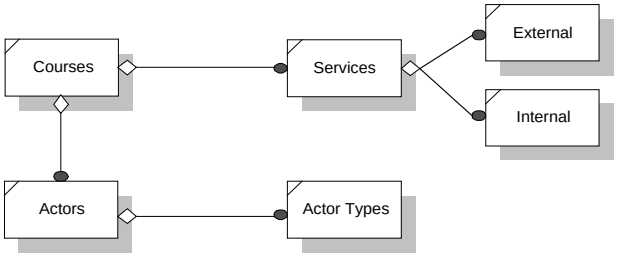

Figure 16. Web-CT class diagram

4.5 LEARNINGSPACE AND VIRTUAL-U

These two environments are put together here because they have similar class structures. Lotus Education and IBM are responsible for the research and development of Lotus LearningSpace (http://www.lotus.com/home.nsf/welcome/learnspace), an educational technology with supporting services for distance education.

The LearningSpace and Virtual-U (http://virtual-u.cs.sfu.ca/vuweb/) class structures are essentially the same as the one presented in Figure 16. However, they additionally introduce the concepts of documents and tasks (Figure 17). The services are based on documents, and each document may have various tasks assigned to it. In LearningSpace documents and tasks can be classified in categories (Figure 18).

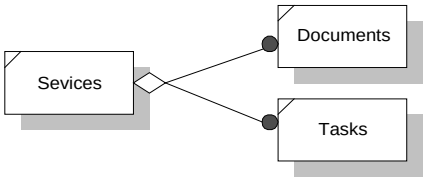

Figure 17. LearningSpace and Virtual-U services class structure

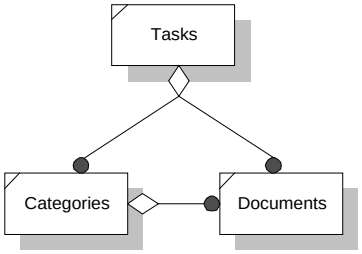

Figure 18. LearningSpace tasks, categories, and documents

4.6 AN OBJECT-ORIENTED FRAMEWORK AND DOMAIN SPECIFIC LANGUAGES FOR WBE

We have tried to capture the core functionality of the analyzed WBE environments to define an object-oriented framework, called ALADIN. We have used the viewpoint unification process to define the framework's kernel structure and it's hot-spots.

Figure 19 illustrates (in an abstract way) how we used each one of the analyzed environments as a viewpoint of the final framework. The basic idea was to identify a kernel that could provide all the functionality required for generating, at least, the six analyzed environments.

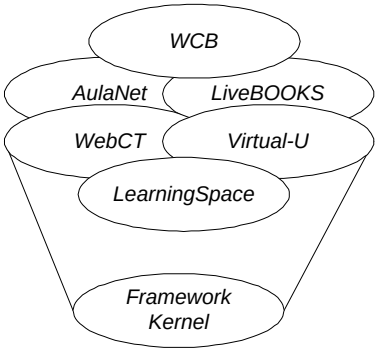

Figure 19. Viewpoints unification (abstractly)

The classes institutions, departments, courses, actor types, and services are responsible for accessing their correspondent information in a database system. To provide this functionality gets and sets methods are present in each of these classes. However the attributes that define these entities vary in each analyzed environment, and therefore all the access methods are marked as hot-pots through the use of the unification rules.

An example of a usual execution of the framework is shown in Figure 20, where the object course asks for the students actors that attend it.

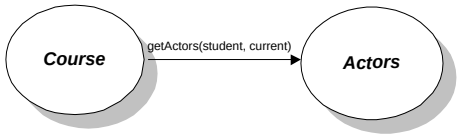

Figure 20. Framework execution example

Since the ALADIN framework allows flexible definitions of the actor's attributes, the method getActors is not defined until the framework is instantiated.

The interface and site classes, used to define the visual representation and navigational structure, are also defined as hot-spots. The interface class depends on the layout structure (usually defined by an interface designer) while the site class may have various implementations to the methods that generate the output HTML files.

Note that the concepts of tasks, documents, and categories that are present in the LearningSpace and Virtual-U environments are not present in the template-hook framework model, although the unification rules define that these classes should be present. This is why these concepts can be implemented through the service class in a more elegant way. This decision was creative, and cannot be formalized.

In the hot-spot instantiation process the separation meta-pattern was selected to implement all the hot-spot relationships in the template-hook framework model. The structure of the final pattern-based framework model generated by the hot-spot instantiation process is now shown in Figures 21, 22, and 23. The signature and implementation for all hook methods will be defined only in the instantiation process, as described in the method.

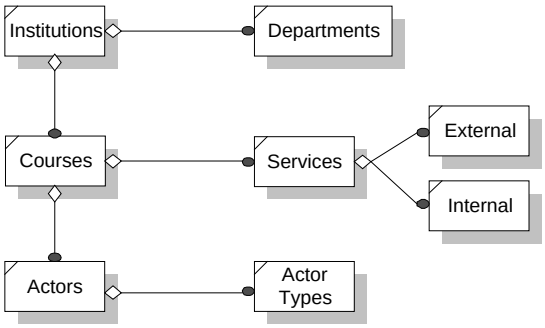

Figure 21. ALADIN OMT class diagram

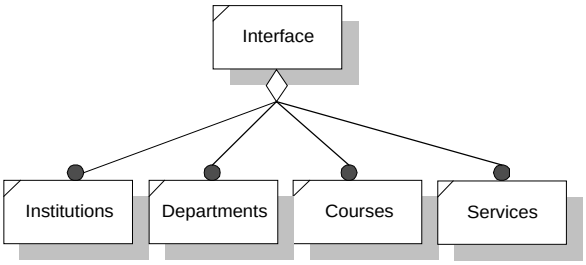

Figure 22. ALADIN interface class structure

ALADIN generates one WWW site for each type of actor defined. Normally two sites are always present in the existent WBE environments: a learning site and an authoring site. However, the ALADIN framework allows that others actor types defined in the system have their own WWW site. As an example we could define a site for the monitors and other site for the secretaries. All the sites generated by the ALADIN framework can define many navigational structures. Figure 23 shows the design representation that defines the navigational structure for the generated sites.

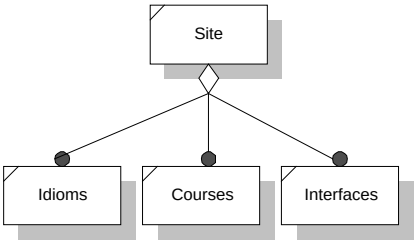

Figure 23. ALADIN site class structure

There are two DSLs used in the ALADIN instantiation process:

Educational Language: used in the definition of the educational components (courses, actors, services, institutions and departments);

Navigational Language: used in the definition of the language (e.g. English, French), interface (e.g. background images, buttons), and navigational structure.

4.6.1 EDUCATIONAL LANGUAGE

The Educational Language code is transformed in two framework instantiation codes: one in Microsoft Access Basic, which is used for defining the database structure that will be used by the environment, and other in Lua and CGILua [13], which provides the access methods for accessing the database. Figure 24 illustrates this architecture. The transformations are made through the use of the Draco-PUC transformational system.

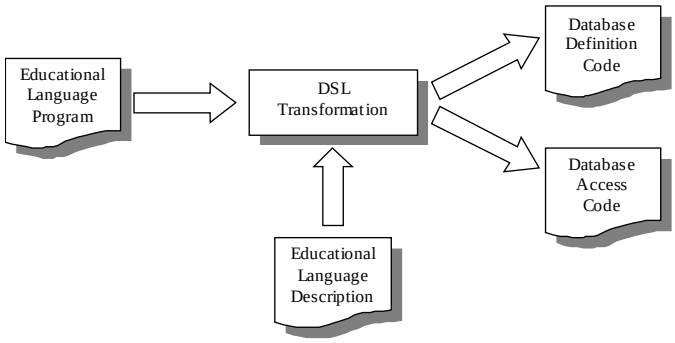

Figure 24. Educational language architecture

The following program code is written in the Educational Language.

```
Institution "PUC-RIO", "Pontificia Universidade
       Catolica - Rio de Janeiro", "PUC.gif";
Department "CETUC", "TELECOMUNICATIONS",
       "CETUC.GIF";
Department "CS", "COMPUTER SCIENCE", "CS.GIF";
Actor Type Teacher, "Teacher"
```

```
        {       name String;
                description Memo;
                Photo Image; };
Actor Type Student, "Student"
        {       name String;
                description Memo;
                period Integer;
                address String;
                average Real; };
Course
        {       name String;
                code String;
                syllabus Memo;
                description Memo;
                image Image; };
Service "CourseNews"
        {       news Memo;
                initialDate Date;
                finalDate Date; }
        read = [ Student ]
        write = [ Teacher ];
```

Note that each language constructor has not only the data used by the environment, but also the meta-information about the structure of these data. The definition of the course attributes in the operator Course is an example.

The execution of the Microsoft Access basic code, generated by the transformation of the above Educational Language program, creates the Microsoft Access database used by the WBE environment that is being instantiated from the ALADIN framework. As described above, the transformation of the Educational Language also generates the database access methods. The following methods interfaces, written in Lua, define the methods generated to access course definitions from the previously shown Educational Language program.

```
luaTable = getCourse(name, code, syllabus,
        description, image)
addCourse (name, code, syllabus, description,
        image)
updateCourse(oldName, name,  code, syllabus,
        description, image)
deleteCourse(name)
```

These methods encapsulate the SQL commands, allowing ALADIN users to generate WBE environments without having any knowledge in manipulating relational databases.

4.6.2 NAVIGATIONAL LANGUAGE

The Navigational Language code is transformed (also using Draco-PUC) into HTML and Lua files that define the WBE environment interface and navigational structure. These files also have the embedded CGILua access methods above described. Therefore, the transformation of the Navigational Language generates the WBE environment site structure.

An example of the Navigational Language is provided next.

```
Language "English"
Language "Portuguese"
Text "title1", " English", "Resources"
Text "title1", "Portuguese", "Recursos"
Image "img1", " English", "c:\ing\img.gif"
Image "img1","Portuguese","c:\port\img.gif"
a := template("c:\templates\temp1.html")
b := template("c:\templates\temp2.html")
c := template("c:\templates\temp3.html")
b.next := c
b.previous := a
```

This language provides an operator for defining the languages supported by the environment (language). The texts and images used by the environment are defined in the various languages. Note that the HTML files are in fact templates, which have special tags for the texts and images. In this example the tag <ALADIN-TEXT>title1</ALADIN-TEXT> would be replaced by the string “Resources”, if the selected language had been English, and by the string “Recursos”, if the selected language was Portuguese.

The same is valid for the hypertext links (navigational structure). In the above example, the tag <ALADIN-LINK>previous</ALADIN-LINK> in the template temp2.html would be replaced by the string <A HREF="c:\templates\temp1.html">previous</A>, while the tag <ALADIN-LINK>next</ALADIN-LINK> would be replaced by the string <A HREF = "c:\templates\temp3.html">next</A>. This approach allows the definition of all the navigational links in a separated file, providing more readability and flexibility.

4.7 ALADIN **UTILIZATION** Example

This section describes how the AulaNet authoring site was developed using the ALADIN framework. The general structure of the authoring site is shown in Figure 25.

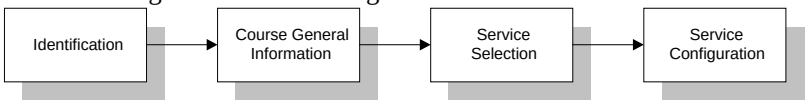

Figure 25. AulaNet authoring site: global view

In the identification step the author has to provide an userid and password to the system. In the course general information step all the basic information about the course must be completed, such as course name, description and syllabus[3]. Then the author has to inform the system about the services necessary for the course. Finally, in the service configuration phase, each internal service available in the course must be configured. Once all these steps have been completed the course is ready and the learning site can be generated. The structure and HTML code (with embedded CGILua) generated by the ALADIN framework for the resource selection step is shown below.

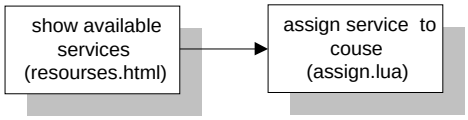

Figure 26. Resource selection: detailed structure

```
**************************************
*             services.html          *
**************************************
<HTML><HEAD>
<TITLE>Show Available Services</TITLE>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000"
      topmargin=0 leftmargin=0>
<H1> Please select the services you want to use in your course </H1>
<FORM NAME="data" TARGET="_top" METHOD="post" ACTION="assign.lua">
<!--$$
-- Select all the services availabe to the
-- current intitution. The result of this
-- selection is stored in a Lua table.
table = getService(currentInstitution)
while (table ~= nil) do
  write('<td valign=middle>')
  write('<INPUT TYPE="Checkbox"
       NAME="'..table[0]..'" VALUE="'..table[0]..'">')
  write('</td>')
  moveNext(table)
end
$$-->
<INPUT TYPE="Submit" VALUE="Next">
</FORM></BODY></HTML>

**************************************
*              assign.lua            *
**************************************
table = getService(currentInstitution)
while (table ~= nil) do
  -- check if the resource is select or not
  -- if true add the resource to the course
  if (cgi.table[0] = "ON") then
       has(course, table[0])
  moveNext(table)
end
```

The generated HTML and Lua files are very simple and easy to be understood since they use high-level methods, which compose the Database Access Code.

[3] The information of which are the fields to be completed in the course general information step is provided by the Course operator (Educational Language).

5. THE DESIGN METHOD FORMALIZATION APPROACH

The framework design method formalization is presented here using the Z specification language [32, 33]. The artifacts meta-model (Figure 27) was defined to precisely describe the object-oriented concepts necessary for the method specification. This meta-model was based in the OMT formalization presented in [37], where all the artifacts necessary for that design method were presented through Z schemas. In this section the meta-model is formally presented and Sections 6 and 7 present the formalization of the two processes: viewpoint unification and hot-spot instantiation.

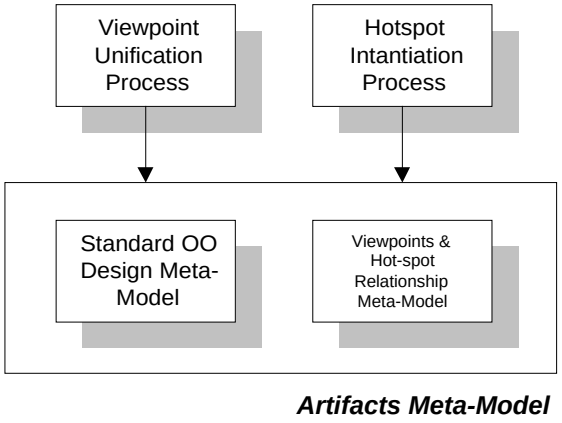

Figure 27. Formalization architecture

Classes, methods, and relationships are modeled as given sets of unelaborated entities in Z.

$$[CLASS, METHOD, RELATIONSHIP]$$

The schema DesignData defines the sets of classes, methods and relationships needed by each one of the artifacts: the set of viewpoints, the template-hook framework model, and the pattern-based framework model.

DesignData

$$
\begin{aligned}
&class: \mathbb{P}\ CLASS \\
&method: \mathbb{P}\ METHOD \\
&relationship: \mathbb{P}\ RELATIONSHIP \\
&vClass: \mathbb{P}\ CLASS \\
&vMethod: \mathbb{P}\ METHOD \\
&vRelationship: \mathbb{P}\ RELATIONSHIP \\
&thClass: \mathbb{P}\ CLASS \\
&thMethod: \mathbb{P}\ METHOD \\
&thRelationship: \mathbb{P}\ RELATIONSHIP \\
&pbClass: \mathbb{P}\ CLASS \\
&pbMethod: \mathbb{P}\ METHOD \\
&pbRelationship: \mathbb{P}\ RELATIONSHIP \\
\hline
&vClass \neq \varnothing \\
&thClass \neq \varnothing \\
&pbClass \neq \varnothing \\
&vClass \cap thClass = \varnothing \\
&vClass \cap pbClass = \varnothing \\
&thClass \cap pbClass = \varnothing \\
&vMethod \cap thMethod = \varnothing \\
&vMethod \cap pbMethod = \varnothing \\
&thMethod \cap pbMethod = \varnothing \\
&vRelationship \cap thRelationship = \varnothing \\
&vRelationship \cap pbRelationship = \varnothing \\
&thRelationship \cap pbRelationship = \varnothing \\
&class = vClass \cup thClass \cup pbClass \\
&method = vMethod \cup thMethod \cup pbMethod \\
&relationship = vRelationship \cup thRelationship \cup pbRelationship
\end{aligned}
$$

Schema DesignData defines:

- a set of all the classes in the meta-model (class);
- a set of all the methods in the meta-model (method);
- a set of all the relationships in the meta-model (relationship);
- a set of all the classes in the set of viewpoints (vClass);
- a set of all the methods in the set of viewpoints (vMethod);
- a set of all the relationships in the set of viewpoints (vRelationship);
- a set of all the classes in the template-hook framework model (thClass);
- a set of all the methods in the template-hook framework model (thMethod);
- a set of all the relationships in the template-hook framework model (thRelationship);
- a set of all the classes in the pattern-based framework model (pbClass);
- a set of all the methods in the pattern-based framework model (pbMethod);
- a set of all the relationships in the pattern-based framework model (pbRelationship);
- a restriction that the number of classes in each of the artifacts is at least one;
- a restriction that all the sets of classes, methods, and relationships are disjoint;
- a definition that the sets class, method, and relationship contain all the classes, methods, and relationships in the meta-model, respectively.

Each class in the model has a unique name. In order to formalize the class names a given is introduced.

```
[CLASS_NAME]
```

```
ClassName
DesignData
className: P CLASS_NAME
nameOfClass: CLASS → CLASS_NAME
---
nameOfClass ∈ class → className
∀ c1, c2: thClass •
       nameOfClass(c1) ≠ nameOfClass(c2)
∀ c1, c2: pbClass •
       nameOfClass(c1) ≠ nameOfClass(c2)
```

Schema ClassName defines:

- a set of class names (className);
- a total surjection function (↠) from classes to class names (nameOfClass);
- a restriction that if two classes belong to the template-hook framework model they must have different names;
- a restriction that if two classes belong to the pattern-based framework model they must have different names.

Each class in the model also has a set of methods, as defined by the following schema.

```
MethodClass
DesignData
classOfMethod: METHOD → CLASS
---
classOfMethod ∈ method → class
```

Schema MethodClass defines:

- a total surjection function from methods to classes (classOfMethod). Note that the specification of this function implies that every class must have at least one method.

Each method has a unique name. In order to formalize it a given set is introduced.

```
[METHOD_NAME]
```

```
MethodName
DesignData
MethodClass
methodName: P METHOD_NAME
nameOfMethod: METHOD → METHOD_NAME
---
nameOfMethod ∈ method → methodName
∀ m1, m2: method |
       classOfMethod(m1) = classOfMethod(m2) •
              nameOfMethod(m1) ≠ nameOfMethod(m2)
```

Schema MethodName defines:

- a set of method names (methodName);
- a total surjection function from methods to method names (nameOfMethod);
- a restriction that if two methods belong to the same class they must have different names.

Each method in the model also has a signature, which specify its parameters and return value and type, and an implementation. In order to formalize these concepts given sets for all possible signatures and implementations are introduced.

```
[SIGNATURE, IMPLEMENTATION]
```

```
MethodSignature
DesignData
methodSignature: P SIGNATURE
signatureOfMethod: METHOD → SIGNATURE
---
signatureOfMethod ∈ method → methodSignature
```

Schema MethodSignature defines:

- a set of method signatures (methodSignature);
- a total surjection function from methods to method signatures (signatureOfMethod).

```
MethodImplementation
DesignData
methodImplementation: P IMPLEMENTATION
implementationOfMethod: METHOD → IMPLEMENTATION
---
implementationOfMethod ∈ method → methodImplementation
```

Schema MethodImplementation defines:

- a set of class implementations (methodImplementation);
- a total surjection function from methods to method implementations (implementationOfMethod).

The set RELATIONSHIP_TYPE defines all the possible types of relationship [26].

```
RELATIONSHIP_TYPE == {Aggregation, Association, Inheritance}
```

```
RelationshipType
DesignData
typeOfRelationship: RELATIONSHIP → RELATIONSHIP_TYPE
dom typeOfRelationship = relationship
```

Schema RelationshipType defines:

- a total function (→) from relationships to types of relationships (typeOfRelationship);
- a restriction that all the relationships in the system must have type.

Each relationship in the model is related to two classes. One class is the source of the relationship while the other is the target.

```
RelationshipSourceTarget
DesignData
sourceOfRelationship: RELATIONSHIP → CLASS
targetOfRelationship: RELATIONSHIP → CLASS
sourceOfRelationship ∈
        vRelationship → vClass ∪
        thRelationship → thClass ∪
        pbRelationship → pbClass
targetOfRelation ∈
        vRelationship → vClass ∪
        thRelationship → thClass ∪
        pbRelationship → pbClass
```

Schema RelationshipSourceTarget defines:

- a total function from relationships to sources of relationships (sourceOfRelationship). The definition of this function implies that the source class must belong to the same artifact to which the relationship belongs;
- a total function from relationships to targets of relationships (targetOfRelationship). The definition of this function implies that the target class must belong to the same artifact to which the relationship belongs;

The given set VIEWPOINT is used to formalize the set of viewpoints used as input for the viewpoint unification process.

```
[VIEWPOINT]
```

```
ViewpointDesignData
DesignData
viewpoint: P VIEWPOINT
viewpoint ≠ ∅
```

Schema ViewpointDesignData defines:

- a set of viewpoints (viewpoint);
- a restriction that the model must have at least one viewpoint.

Each class in the model belongs to a viewpoint, as described by the ClassViewpoint schema.

```
ClassViewpoint
DesignData
ClassName
ViewpointDesignData
viewpointOfClass: CLASS ⤔ VIEWPOINT
---
viewpointOfClass ∈ vClass → viewpoint
∀ c1, c2: vClass |
        nameOfClass(c1) = nameOfClass(c2) •
                viewpointOfClass(c1) ≠ viewpointOfClass(c2)
```

Schema ClassViewpoint defines:

- a total surjection function from viewpoint classes to viewpoints (viewpointOfClass). Note that the relation between CLASS and VIEWPOINT is only a partial surjection function (⤔), since not all the classes in the model have a viewpoint, just the ones that belong to the set vClass;
- a restriction that if two classes belong to the same viewpoint they must have different names.

As the function viewpointOfClass is a total surjection, every viewpoint must have at least one class.

Like the classes, each relationship in the model also belongs to a unique viewpoint, as described by the RelationshipViewpoint schema.

```
RelationshipViewpoint
DesignData
RelationshipSourceTarget
ViewpointDesignData
ClassViewpoint
viewpointOfRelationship: RELATIONSHIP ↦ VIEWPOINT
---
viewpointOfRelationship ∈ vRelationship → viewpoint
∀ r: vRelationship; c: vClass |
        sourceOfRelationship(r) = c •
                viewpointOfRelationship(r) = viewpointOfClass(c)
∀ r: vRelationship; c: vClass |
        targetOfRelationship(r) = c •
                viewpointOfRelationship(r) = viewpointOfClass(c)
```

Schema RelationshipViewpoint defines:

- a total function from viewpoint relationships to viewpoints (viewpointOfRelationship). Note that the relation between RELATIONSHIP and VIEWPOINT is only a partial function (↦), since not all the relationships in the model have a viewpoint, just the ones that belong to the set vRelationship;
- a restriction that if a relationship belongs to the a viewpoint, their source and target classes must belong to the same viewpoint.

As the result of the viewpoint unification process is a framework model that utilizes the hot-spot relationship (template-hook framework model), this kind of relationship must be formally defined, so a given set for all possible hot-spot relationships is introduced.

```
[HOT_SPOT_RELATIONSHIP]
```

Each hot-spot relationship, like the others relationships previously formalized, has a source and a target class as defined by the HotSpotSourceTarget schema.

HotSpotSourceTarget

```
DesignData
hsRelationship: P HOT_SPOT_RELATIONSHIP
sourceOfHotSpot: HOT_SPOT_RELATIONSHIP → CLASS
targetOfHotSpot: HOT_SPOT_RELATIONSHIP → CLASS
---
sourceOfHotSpot ∈ hsRelationship → thClass
targetOfHotSpot ∈ hsRelationship → thClass
```

Schema HotSpotSourceTarget defines:

- a set of hot-spot relationships (hsRelationships);
- a total function from hot-spot relationships to sources of hot-spot relationships (sourceOfHotSpot);
- a total function from hot-spot relationships to targets of hot-spot relationships (targetOfHotSpot);

6. THE VIEWPOINT UNIFICATION PROCESS SPECIFICATION

This section presents Z schemas that describe the set of consistent viewpoints and the viewpoint unification process. It also proves that the semantics of the template-hook framework model is the same of the initial set of viewpoints.

ConsistentViewpoints

```
DesignData
ClassName
MethodClass
MethodName
MethodSignature
MethodImplementation
RelationshipType
RelationshipSourceTarget
ViewpointDesignData
ClassViewpoint
RelationshipViewpoint
---
∀ r1, r2: vRelationship; c1, c2, c3, c4: vClass |
       sourceOfRelationship(r1) = c1 ∧
       targetOfRelationship(r1) = c2 ∧
       sourceOfRelationship(r2) = c3 ∧
       targetOfRelationship(r2) = c4 ∧
       nameOfClass(c1) = nameOfClass(c3) ∧
       nameOfClass(c2) = nameOfClass(c4)
              • typeOfRelationship (r1) = typeOfRelationship (r2)
```

The schema ConsistentViewpoints defines the set of consistent viewpoints by the restriction that the relationships between classes with same names, in different viewpoints, must have the same type.

To improve the specification readability the viewpoint unification process specification was divided in several schemas: one that specifies the artifacts used in the process and one for each of the unification rules.

ViewpointUnificationBasic

```
ConsistentViewpoints
HotSpotSourceTarget
templateMethod: P METHOD
hookMethod: P METHOD
---
templateMethod ⊆ thMethod
hookMethod ⊆ thMethod
```

Schema ViewpointUnificationBasic defines the artifacts used in the viewpoint unification process:

- the set of consistent viewpoints (ConsistentViewpoints);
- the pattern-based framework model (composed by the schemas defined in the ConsistentViewpoints schema and by the schema HotSpotSourceTarget);
- a set of template methods (templateMethod);
- a set of hook methods (hookMethod);
- a restriction that the template and hook methods are sub-sets of the set thMethod, which contains all the methods present in the template-hook framework model.

Rule1

ViewpointUnificationBasic

```
∀ vc: vClass •
        ∃ thc: thClass |
                nameOfClass(thc) = nameOfClass(vc)
```

Rule2

ViewpointUnificationBasic

```
∀ vm1: vMethod |
        ¬∃ vm2: vMethod |
                nameOfMethod(vm1) = nameOfMethod(vm2) ∧
                nameOfClass(classOfMethod(vm1)) = nameOfClass(classOfMethod(vm2)) ∧
                (signatureOfMethod(vm1) ≠ signatureOfMethod(vm2) ∨
                implementationOfMethod(vm1) ≠ implementationOfMethod(vm2)) •
                        ∃ thc: thClass; thm: thMethod |
                                nameOfClass(thc) = nameOfClass(classOfMethod(vm1)) ∧
                                nameOfMethod(thm) = nameOfMethod(vm1) ∧
                                signatureOfMethod(thm) = signatureOfMethod(vm1) ∧
                                implementationOfMethod(thm) = implementationOfMethod(vm1) ∧
                                classOfMethod(thm) = thc
```

Rule 1 says that every class that belongs to the set of viewpoints has a corresponding class, with same name, in the template-hook framework model. Schema Rule1 defines that stating that for every class (vClass) in the set of viewpoints there is one corresponding class (thClass) in the template-hook framework model that has the same name.

Schema Rule2 defines that:

- if a method (vm1) has the same signature and implementation in all the viewpoints it appear it has a corresponding method (thm), with same name, signature, and implementation, in the template-hook framework model.

Rule3

ViewpointUnificationBasic

```
∀ vm1: vMethod |
        ∃ vm2: vMethod |
                nameOfMethod(vm1) = nameOfMethod(vm2) ∧
                nameOfClass(classOfMethod(vm1)) = nameOfClass(classOfMethod(vm2)) ∧
                signatureOfMethod(vm1) ≠ signatureOfMethod(vm2) •
                        ∃ thc: thClass; thm: thMethod |
                                nameOfClass(thc) = nameOfClass(classOfMethod(vm1)) ∧
                                nameOfMethod(thm) = nameOfMethod(vm1) ∧
                                signatureOfMethod(thm) = UNDEFINED ∧
                                implementationOfMethod(thm) = UNDEFINED ∧
                                classOfMethod(thm) = thc ∧
                                thm ∈ hookMethod
```

Schema Rule3 defines that:

- if a method exists in more then one viewpoint with different signature (vm1 and vm2) it has a corresponding hook method (thm), which belongs to the set hookMethod, in the template-hook framework model, with same name but UNDEFINED signature and implementation.

Rule4

ViewpointUnificationBasic

```
∀ vm1: vMethod |
       ∃ vm2: vMethod |
              nameOfMethod(vm1) = nameOfMethod(vm2) ∧
              nameOfClass(classOfMethod(vm1)) = nameOfClass(classOfMethod(vm2)) ∧
              signatureOfMethod(vm1) = signatureOfMethod(vm2) ∧
              implementationOfMethod(vm1) ≠ implementationOfMethod(vm2) •
                     ∃ thc: thClass; thm: thMethod |
                            nameOfClass(thc) = nameOfClass(classOfMethod(vm1)) ∧
                            nameOfMethod(thm) = nameOfMethod(vm1) ∧
                            signatureOfMethod(thm) = signatureOfMethod(vm1) ∧
                            implementationOfMethod(thm) = UNDEFINED ∧
                            classOfMethod(thm) = thc ∧
                            thm ∈ hookMethod
```

Schema Rule4 defines that:

- if a method exists in more then one viewpoint with different implementation (vm1 and vm2) it has a corresponding hook method (thm), which belongs to the set hookMethod, in the template-hook framework model, with same name and signature but UNDEFINED implementation.

Rule5

ViewpointUnificationBasic

```
∀ vm1, vm2: vMethod; vr: vRelationship |
       (vm1, vr, vm2) ∈ uses ∧
       (∃ hm: thMethod |
              hm ∈ hookMethod ∧
              nameOfMethod(vm2) = nameOfMethod(hm) ∧
              nameOfClass(classOfMethod(vm2)) = nameOfClass(classOfMethod(hm))) •
                     ∃ tc: thClass; tm: thMethod; hsr: hsRelationship |
                            nameOfClass(tc) = nameOfClass(classOfMethod(vm1)) ∧
                            nameOfMethod(tm) = nameOfMethod(vm1) ∧
                            classOfMethod(tm) = tc ∧
                            tm ∈ templateMethod ∧
                            sourceOfHotSpot(hsr) = tc ∧
                            targetOfHotSpot(hsr) = classOfMethod(hm)
```

Schema Rule5 defines that:

- all the methods that uses hook methods (hm) are defined as template methods (tm). For guarantying that a relation between viewpoint classes and viewpoint relationships is defined. This relation, called uses, defines that if (c1, r, c2) belongs to the relation class c1 uses class c2 through the relation r. Note that if the template and hook methods belong to the same class the relationship will be UNDEFINED in the relation uses, e.g. (template, UNDEFINED, hook) ;
- there is always a hot-spot relationship (hsr) between the class that has the template method (tc) and the class that has its correspondent hook method (classOfMethod(hm)).

Rule6

ViewpointUnificationBasic

```
∀ vm1, vm2: vMethod; vr: vRelationship |
              (vm1, vr, vm2) ∈ uses  ∧
              (¬∃ hm: thMethod |
                     hm ∈ hookMethod ∧
                     nameOfMethod(vm2) = nameOfMethod(hm) ∧
                     nameOfClass(classOfMethod(vm2)) = nameOfClass(classOfMethod(hm))) •
                 ∃ thr: thRelationship |
                     nameOfClass(sourceOfHotSpot(thr)) = nameOfClass(classOfMethod(vm1)) ∧
                     nameOfClass(targetOfHotSpot(thr)) = nameOfClass(classOfMethod(vm2)) ∧
                     typeOfRelationship(thr) = typeOfRelationship(vr)

∀ vm1, vm2: vMethod; vr: vRelationship |
              (vm1, vr, vm2) ∈ uses  ∧
              typeOfRelationship(vr) = INHERITANCE •
                 ∃ thr: thRelationship |
                     nameOfClass(sourceOfHotSpot(thr)) = nameOfClass(classOfMethod(vm1)) ∧
                     nameOfClass(targetOfHotSpot(thr)) = nameOfClass(classOfMethod(vm2)) ∧
                     typeOfRelationship(thr) = INHERITANCE
```

Schema Rule6 defines that:

- all the existing relationships (vr) in the set of viewpoints that do not have a corresponding hot-spot relationship are maintained in the in the template-hook framework model (thr);
- all the INHERITANCE relationships are maintained, since they can never be transformed into hot-spot relationships. This is because inheritance relationships can not be used to access methods (like association and aggregation). For this reason none of the relationships that belong to the relation uses is an inheritance relationship.

The ViewpointUnificationProcess schema can be defined as the conjunction of all the rules schemas, as shown bellow.

```
ViewpointUnificationProcess ≅ Rule1 ∧ Rule2 ∧ Rule3 ∧ Rule4 ∧ Rule5 ∧ Rule6
```

Soundness of the Viewpoint Unification Process

Since the process is completely specified it is possible to prove that it is correct by showing that the generated template-hook framework model has the same semantics of the initial set of viewpoints. To prove that it is necessary to prove that all the methods, relationships and classes present in the initial set of viewpoints are also present in the template-hook framework model.

Rule1 assures this correspondence is valid for the classes. Let us suppose that there is a class vc in the initial set of viewpoints that do not belong to the template-hook framework model.

```
∃ vc: vClass | (¬∃ thc: thClass | nameOfClass(vc) = nameOfClass(thc))
```

Rule1 says that

```
∀ vc: vClass • ∃ thc: thClass | nameOfClass(thc) = nameOfClass(vc)
```

Which happens that

```
(¬∃ thc: thClass | nameOfClass(vc) = nameOfClass(thc)) ∧
(∃ thc: thClass | nameOfClass(vc) = nameOfClass(thc))
```

That is a contradiction, and thus the class correspondence is proved to be valid. Rules 2, 3, and 4 assure the method correspondence. Let us suppose that there is a method vm in the initial set of viewpoints that do not belong to the template-hook framework model.

```
∃ vm1: vMethod | (¬∃ thm: thMethod | nameOfMethod(vm1) = nameOfMethod(thm))
```

Since vm1 belongs to the initial set of viewpoints there are only there options that can occur to it:

1. Rule2: there is no other method (vm2) in the set of viewpoints with same name but different signature or implementation:

```
¬∃ vm2: vMethod |
        nameOfMethod(vm1) = nameOfMethod(vm2) ∧
        nameOfClass(classOfMethod(vm1)) = nameOfClass(classOfMethod(vm2)) ∧
        (signatureOfMethod(vm1) ≠ signatureOfMethod(vm2) ∨
        implementationOfMethod(vm1) ≠ implementationOfMethod(vm2))
```

2. Rule3: there is other method (vm2) in the set of viewpoints with same name but different signature:

```
∃ vm2: vMethod |
        nameOfMethod(vm1) = nameOfMethod(vm2) ∧
        nameOfClass(classOfMethod(vm1)) = nameOfClass(classOfMethod(vm2)) ∧
        signatureOfMethod(vm1) ≠ signatureOfMethod(vm2)
```

3. Rule4: there is other method (vm2) in the set of viewpoints with same name and signature but different implementation:

```
∃ vm2: vMethod |
        nameOfMethod(vm1) = nameOfMethod(vm2) ∧
        nameOfClass(classOfMethod(vm1)) = nameOfClass(classOfMethod(vm2)) ∧
        signatureOfMethod(vm1) = signatureOfMethod(vm2) ∧
        implementationOfMethod(vm1) ≠ implementationOfMethod(vm2)
```

But all the three schemas (Rule2, Rule3, and Rule4) state that:

```
∃ thm: thMethod | nameOfMethod(thm) = nameOfMethod(vm1)
```

Since one of the three schemas is always valid, it happens that

```
(¬∃ thm: thMethod | nameOfMethod(vm1) = nameOfMethod(thm)) ∧
(∃ thm: thMethod | nameOfMethod(thm) = nameOfMethod(vm1))
```

That is a contradiction, and thus the method correspondence is proved to be valid. Rules 5 and 6 assures the relationship correspondence. Since the association and aggregation relationships function is to allow classes to use methods defined in other classes, all the association and aggregation relationships present in the initial set of viewpoints are also present in the uses relation.

```
∀ vr: vRelationship | typeOfRelationship(vr) = ASSOCIATION  ∨
                      typeOfRelationship(vr) = AGGREGATION •
                              ∃ vc1, vc2: vClass | (vc1, vr, vc2)
```

Let us suppose that there is a relationship vr (association or aggregation) in the initial set of viewpoints that do not belong to the template-hook framework model.

```
∃ vr: vRelationship |
      (typeOfRelationship(vr) = ASSOCIATION  ∨
      typeOfRelationship(vr) = AGGREGATION) ∧
      (¬∃ thr: thRelationship |
         nameOfClass(sourceOfRelationship(vr)) = nameOfClass(sourceOfRelationship(thr)) ∧
         nameOfClass(targetOfRelationship(vr)) = nameOfClass(targetOfRelationship(thr))) ∧
      (¬∃ hsr: hsRelationship |
         nameOfClass(sourceOfRelationship(vr)) = nameOfClass(sourceOfHotSpot(hsr)) ∧
         nameOfClass(targetOfRelationship(vr)) = nameOfClass(targetOfHotSpot(hsr)))
```

Since vr belongs to the uses relation there are only two options that can occur to it:

1. Rule5: it is transformed in to a hot-spot relationship in the template-hook framework model.

```
∀ vm1, vm2: vMethod; vr: vRelationship |
        (vm1, vr, vm2) ∈ uses  ∧
        (∃ hm: thMethod  |
                hm ∈ hookMethod ∧
                nameOfMethod(vm2) = nameOfMethod(hm) ∧
                nameOfClass(classOfMethod(vm2)) = nameOfClass(classOfMethod(hm)))
```

2. Rule6: its relationship type is maintained in the template-hook framework model:

```
∀ vm1, vm2: vMethod; vr: vRelationship |
        (vm1, vr, vm2) ∈ uses  ∧
        (¬∃ hm: thMethod |
                hm ∈ hookMethod ∧
                nameOfMethod(vm2) = nameOfMethod(hm) ∧
                nameOfClass(classOfMethod(vm2)) = nameOfClass(classOfMethod(hm)))
```

But schema Rule5 states that:

```
∃ tc: thClass; tm: thMethod; hsr: hsRelationship |
        nameOfClass(tc) = nameOfClass(classOfMethod(vm1)) ∧
        nameOfMethod(tm) = nameOfMethod(vm1) ∧
        classOfMethod(tm) = tc  ∧
        tm ∈ templateMethod  ∧
        sourceOfHotSpot(hsr) = tc ∧
        targetOfHotSpot(hsr) = classOfMethod(hm)
```

Which implies that:

```
∃ hsr: hsRelationship |
        nameOfClass(sourceOfHotSpot(hsr)) = nameOfClass(sourceOfRelationship(vr))∧
        nameOfClass(targetOfHotSpot(hsr)) = nameOfClass(targetOfRelationship(vr))
```

And schema Rule6 states that:

```
∃ thr: thRelationship |
        nameOfClass(sourceOfHotSpot(thr)) = nameOfClass(classOfMethod(vm1)) ∧
        nameOfClass(targetOfHotSpot(thr)) = nameOfClass(classOfMethod(vm2)) ∧
```

Which implies that:

```
∃ thr: thRelationship |
        nameOfClass(sourceOfHotSpot(thr)) = nameOfClass(sourceOfRelationship(vr)) ∧
        nameOfClass(targetOfHotSpot(thr)) = nameOfClass((targetOfRelationship(vr))
```

Since always one of the two schemas will be valid, we have that:

```
∃ vr: vRelationship |
        (typeOfRelationship(vr) = ASSOCIATION  ∨
        typeOfRelationship(vr) = AGGREGATION) ∧
        (¬∃ thr: thRelationship |
           nameOfClass(sourceOfRelationship(vr)) = nameOfClass(sourceOfRelationship(thr)) ∧
           nameOfClass(targetOfRelationship(vr)) = nameOfClass(targetOfRelationship(thr))) ∧
        (¬∃ hsr: hsRelationship |
           nameOfClass(sourceOfRelationship(vr)) = nameOfClass(sourceOfHotSpot(hsr)) ∧
           nameOfClass(targetOfRelationship(vr)) = nameOfClass(targetOfHotSpot(hsr))) ∧
```

```
((∃ hsr: hsRelationship |
   nameOfClass(sourceOfRelationship(vr)) = nameOfClass(sourceOfHotSpot(hsr)) ∧
   nameOfClass(targetOfRelationship(vr)) = nameOfClass(targetOfHotSpot(hsr))) ∨
(∃ thr: thRelationship |
        nameOfClass(sourceOfHotSpot(thr)) = nameOfClass(sourceOfRelationship(vr)) ∧
        nameOfClass(targetOfHotSpot(thr)) = nameOfClass((targetOfRelationship(vr))))
```

That is a contradiction, and thus the method correspondence is proved to be valid for the all the association and aggregation relationships. For the inheritance relationship the correspondence can be proved through the use of rule 6. Let us suppose that there is an inheritance relationship vr in the initial set of viewpoints that do not belong to the template-hook framework model.

```
∃ vr: vRelationship |
   typeOfRelationship(vr) = INHERITANCE ∧
   (¬∃ thr: thRelationship |
      nameOfClass(sourceOfRelationship(vr)) = nameOfClass(sourceOfRelationship(thr)) ∧
      nameOfClass(targetOfRelationship(vr)) = nameOfClass(targetOfRelationship(thr)))
```

But schema Rule6 states that:

```
∀ vm1, vm2: vMethod; vr: vRelationship |
        typeOfRelationship(vr) = INHERITANCE •
          ∃ thr: thRelationship |
               nameOfClass(sourceOfHotSpot(thr)) = nameOfClass(classOfMethod(vm1)) ∧
               nameOfClass(targetOfHotSpot(thr)) = nameOfClass(classOfMethod(vm2)) ∧
               typeOfRelationship(thr) = INHERITANCE
```

Which implies that:

```
∃ thr: thRelationship |
        nameOfClass(sourceOfHotSpot(thr)) = nameOfClass(sourceOfRelationship(vr)) ∧
        nameOfClass(targetOfHotSpot(thr)) = nameOfClass((targetOfRelationship(vr))
```

So we have that:

```
∃ vr: vRelationship |
   typeOfRelationship(vr) = INHERITANCE ∧
    (¬∃ thr: thRelationship |
      nameOfClass(sourceOfRelationship(vr)) = nameOfClass(sourceOfRelationship(thr)) ∧
      nameOfClass(targetOfRelationship(vr)) = nameOfClass(targetOfRelationship(thr))) ∧
    (∃ thr: thRelationship |
        nameOfClass(sourceOfHotSpot(thr)) = nameOfClass(sourceOfRelationship(vr)) ∧
        nameOfClass(targetOfHotSpot(thr)) = nameOfClass((targetOfRelationship(vr)))
```

That is a contradiction, and thus the method correspondence is proved to be valid for the all the inheritance relationships. Since the correspondence is proved for all the classes, methods, and relationships the viewpoint unification process is proved to be correct.

7. THE HOT-SPOT INSTANTIATION PROCESS SPECIFICATION

This section presents Z schemas that describe the hot-spot instantiation process. It also proves that the method assures the design properties of loose coupling and design complexity. To improve the specification readability the hot-spot instantiation process specification was divided in several schemas:

- one for each combination of the two flexibility properties, this means that a different schema is defined for each different row present in Table 1;

- one to handle the class correspondence between the template-hook and the pattern-based framework models;
- one to handle the method correspondence between the template-hook and the pattern-based framework models;
- one to handle the relationship correspondence between the template-hook and the pattern-based framework models.

Note that the schema ViewpointUnificationProcess is included in all the schemas in this section since it uses all the artifacts needed by the hot-spot instantiation process.

Unification

ViewpointUnificationProcess

```
∀ hsr: hsRelationship; tm, hm: thMethod |
       nameOfClass(sourceOfHotSpot(hsr)) = nameOfClass(classOfMethod (tm)) ∧
       nameOfClass(targetOfHotSpot(hsr)) = nameOfClass(classOfMethod (hm)) ∧
       designPattern(hsr) = UNIFICATION  •
              ∃ pbc: pbClass; pbm1, pbm2: pbMethod |
                     nameOfClass(pbc) = nameOfClass(sourceOfHotSpot(hsr)) ∧
                     classOfMethod(pbm1)= pbc  ∧
                     classOfMethod(pbm2)= pbc  ∧
                     nameOfMethod(pbm1) = nameOfMethod(tm) ∧
                     nameOfMethod(pbm2) = nameOfMethod(hm)
```

Schema Unification defines that:

- if a hot-spot relationship (hrs) has to be implemented through the use of the unification meta-pattern (designPattern(hsr) = UNIFICATION) the template (pbm1) and hook (pbm2) methods belong to the same class (pbc) in the pattern-based framework model.

Separation

ViewpointUnificationProcess

```
∀ hsr: hsRelationship; tm, hm: thMethod |
       nameOfClass(sourceOfHotSpot(hsr)) = nameOfClass(classOfMethod (tm)) ∧
       nameOfClass(targetOfHotSpot(hsr)) = nameOfClass(classOfMethod (hm)) ∧
       classOfMethod (tm) ≠ classOfMethod (hm) ∧
       designPattern(hsr) = SEPARATION •
              ∃ pbc1, pbc2: pbClass; pbm1, pbm2: pbMethod; pbr: pbRelationship |
                     nameOfClass(pbc1) = nameOfClass(sourceOfHotSpot(hsr)) ∧
                     nameOfClass(pbc2) = nameOfClass(targetOfHotSpot(hsr)) ∧
                     classOfMethod(pbm1)= pbc1  ∧
                     classOfMethod(pbm2)= pbc2  ∧
                     nameOfMethod(pbm1) = nameOfMethod(tm) ∧
                     nameOfMethod(pbm2) = nameOfMethod(hm) ∧
                     sourceOfRelationship(pbr) = pbc1 ∧
                     targetOfRelationship(pbr) = pbc2 ∧
                     typeOfRelationship(pbr) = AGGREGATION

 ∀ hsr: hsRelationship; tm, hm: thMethod |
       nameOfClass(sourceOfHotSpot(hsr)) = nameOfClass(classOfMethod (tm)) ∧
       nameOfClass(targetOfHotSpot(hsr)) = nameOfClass(classOfMethod (hm)) ∧
       classOfMethod (tm) = classOfMethod (hm) ∧
       designPattern(hsr) = SEPARATION •
              ∃ pbc1, pbc2: pbClass; pbm1, pbm2: pbMethod; pbr: pbRelationship |
                     nameOfClass(pbc1) = nameOfClass(sourceOfHotSpot(hsr)) ∧
                     nameOfClass(pbc2) = concat(nameOfMethod(hm), "Class") ∧
                     classOfMethod(pbm1)= pbc1  ∧
                     classOfMethod(pbm2)= pbc2  ∧
                     nameOfMethod(pbm1) = nameOfMethod(tm) ∧
                     nameOfMethod(pbm2) = nameOfMethod(hm) ∧
                     sourceOfRelationship(pbr) = pbc1 ∧
                     targetOfRelationship(pbr) = pbc2 ∧
                     typeOfRelationship(pbr) = AGGREGATION
```

Schema Separation defines that:

- if a hot-spot relationship (hrs) has to be implemented through the use of the separation meta-pattern (designPattern(hsr) = SEPARATION) the template (pbm1) and hook (pbm2) methods belong to the separate classes (pbc1 and pbc2) in the pattern-based framework model, which are related through an aggregation relationship (pbr);
- note that if the hot-spot relationship (hsr) has the same source and target classes a new class has to be defined in the pattern-based framework model to hold the hook method. This class is defined with same name of the hook method concatenated with the string "Class". The concat function is not defined here but can be easily implemented through any programming language.

RecursiveUnification

ViewpointUnificationProcess

```
∀ hsr: hsRelationship; tm, hm: thMethod |
       nameOfClass(sourceOfHotSpot(hsr)) = nameOfClass(classOfMethod (tm)) ∧
       nameOfClass(targetOfHotSpot(hsr)) = nameOfClass(classOfMethod (hm)) ∧
       designPattern(hsr) = RECURSIVE_UNIFICATION •
              ∃ pbc: pbClass; pbm1, pbm2: pbMethod; pbr: pbRelationship |
                     nameOfClass(pbc) = nameOfClass(sourceOfHotSpot(hsr)) ∧
                     classOfMethod(pbm1)= pbc  ∧
                     classOfMethod(pbm2)= pbc  ∧
                     nameOfMethod(pbm1) = nameOfMethod(tm) ∧
                     nameOfMethod(pbm2) = nameOfMethod(hm) ∧
                     sourceOfRelationship(pbr) = pbc ∧
                     targetOfRelationship(pbr) = pbc ∧
                     typeOfRelationship(pbr) = AGGREGATION
```

Schema RecursiveUnification defines that:

- if a hot-spot relationship (hrs) has to be implemented through the use of the recursive unification meta-pattern (designPattern(hsr) = RECURSIVE_UNIFICATION) the template (pbm1) and hook (pbm2) methods belong to the same class (pbc) in the pattern-based framework model, and there is an aggregation relationship between this class and itself.

Schema RecursiveSeparation defines that:

- if a hot-spot relationship (hrs) has to be implemented through the use of the separation meta-pattern (designPattern(hsr) = RECURSIVE_SEPARATION) the template (pbm1) and hook (pbm2) methods belong to the separate classes (pbc1 and pbc2) in the pattern-based framework model, which are related through an aggregation relationship (pbr1). There is also an inheritance relationship (pbr2) between the template (pbc1) and hook (pbc2) classes;
- like in the Recursive schema, if the hot-spot relationship (hsr) has the same source and target classes a new class has to be defined in the pattern-based framework model to hold the hook method.

RecursiveSeparation

ViewpointUnificationProcess

```
∀ hsr: hsRelationship; tm, hm: thMethod |
       nameOfClass(sourceOfHotSpot(hsr)) = nameOfClass(classOfMethod (tm)) ∧
       nameOfClass(targetOfHotSpot(hsr)) = nameOfClass(classOfMethod (hm)) ∧
       classOfMethod (tm) ≠ classOfMethod (hm) ∧
       designPattern(hsr) = RECURSIVE_SEPARATION •
              ∃ pbc1, pbc2: pbClass; pbm1, pbm2: pbMethod; pbr1, pbr2: pbRelationship |
                     nameOfClass(pbc1) = nameOfClass(sourceOfHotSpot(hsr)) ∧
                     nameOfClass(pbc2) = nameOfClass(targetOfHotSpot(hsr)) ∧
                     classOfMethod(pbm1)= pbc1  ∧
                     classOfMethod(pbm2)= pbc2  ∧
                     nameOfMethod(pbm1) = nameOfMethod(tm) ∧
                     nameOfMethod(pbm2) = nameOfMethod(hm) ∧
                     sourceOfRelationship(pbr1) = pbc1 ∧
                     targetOfRelationship(pbr1) = pbc2 ∧
                     typeOfRelationship(pbr1) = AGGREGATION ∧
                     sourceOfRelationship(pbr2) = pbc1 ∧
                     targetOfRelationship(pbr2) = pbc2 ∧
                     typeOfRelationship(pbr2) = INHERITANCE

 ∀ hsr: hsRelationship; tm, hm: thMethod |
       nameOfClass(sourceOfHotSpot(hsr)) = nameOfClass(classOfMethod (tm)) ∧
       nameOfClass(targetOfHotSpot(hsr)) = nameOfClass(classOfMethod (hm)) ∧
       classOfMethod (tm) = classOfMethod (hm) ∧
       designPattern(hsr) = RECURSIVE_SEPARATION •
              ∃ pbc1, pbc2: pbClass; pbm1, pbm2: pbMethod; pbr1, pbr2: pbRelationship |
                     nameOfClass(pbc1) = nameOfClass(sourceOfHotSpot(hsr)) ∧
                     nameOfClass(pbc2) = concat(nameOfMethod(hm), “Class”) ∧
                     classOfMethod(pbm1)= pbc1  ∧
                     classOfMethod(pbm2)= pbc2  ∧
                     nameOfMethod(pbm1) = nameOfMethod(tm) ∧
                     nameOfMethod(pbm2) = nameOfMethod(hm) ∧
                     sourceOfRelationship(pbr1) = pbc1 ∧
                     targetOfRelationship(pbr1) = pbc2 ∧
                     typeOfRelationship(pbr1) = AGGREGATION
                     sourceOfRelationship(pbr2) = pbc1 ∧
                     targetOfRelationship(pbr2) = pbc2 ∧
                     typeOfRelationship(pbr2) = INHERITANCE
```

ClassCorrespondence

ViewpointUnificationBasic

```
∀ thc: thClass |
       ¬ ∃ thm1: thMethod; hsr: hsRelationship |
              thm1 ∈ hookClass  ∧
              classOfMethod(thm1) = thc  ∧
              (¬ ∃ thm2: thMethod | thm1 ≠ thm2 ∧ classOfMethod(thm2) = thc)  ∧
              targetOfHotSpot(hsr) = thc  ∧
              (designPattern(hsr) ≠ UNIFICATION  ∨
              designPattern(hsr) ≠ RECURSIVE_UNIFICATION) •
                     ∃ pbc: pbClass |
                            nameOfClass(pbc) = nameOfClass(thc)
```

Schema ClassCorrespondence defines that:

- for every class (thc) in the template-hook framework model there is a correspondent class (pbc) in the pattern-based framework model, except in the cases where the class is a hook class, with only one method (that is the hook method) and that for every hot-spot relationship that uses it the unification patterns are generated.

MethodCorrespondence

ViewpointUnificationBasic

```
∀ thm: thMethod •
       ∃ pbm: pbMethod |
              nameOfMethod(pbm) = nameOfMethod(thm) ∧
              signatureOfMethod(pbm) = signatureOfMethod(thm) ∧
              implementationOfMethod(pbm) = implementationOfMethod(thm)

∀ thm: thMethod | thm ∉ templateMethod ∧ thm ∉ hookMethod
       ∃ pbm: pbMethod |
              nameOfClass(classOfMethod(pbm)) = nameOfClass(classOfMethod(thm))
```

Schema MethodCorrespondence defines that:

- for every method in the template-hook framework model (thm) there is a correspondent method in the pattern-based framework model (pbm), with same name, signature, and implementation;
- for every method in the template-hook framework model that is nor a template neither a hook method its containing class in the in the pattern-based framework model does not chance. Note that the definition of classes for the template and hook methods is provided by the meta-pattern schemas (Unification, Separation, RecursiveUnification, and RecursiveSeparation.

RelationshipCorrespondence

ViewpointUnificationProcess

```
∀ thr: thRelationship •
       ∃ pbr: pbRelationship |
              nameOfClass(sourceOfRelationship(pbr)) =
                     nameOfClass(sourceOfRelationship(thr)) ∧
              nameOfClass(targetOfRelationship(pbr)) =
                     nameOfClass(targetOfRelationship(thr))
```

Schema RelationshipCorrespondence defines that:

- for every relationship in the template-hook framework model (thr) there is a correspondent relationship in the pattern-based framework model (pbr).

The HotSpotInstantiationProcess schema can be defined as the conjunction of all the rules schemas, as shown bellow.

```
HotSpotInstantiationProcess ≅ Unification ∧ Separation ∧ RecursiveUnification
                                ∧ RecursiveSeparation ∧ ClassCorrespondence
                                ∧ MethodCorrespondence ∧ RelationshipCorrespondence
```

Proof of the Design Loose Coupling Property

The separation meta-patterns lead to the design loose coupling, where the template and hot-spot classes are disjoint. To prove this property let us suppose that a hot-spot relationship is implemented through a separation meta-pattern and that the template and hook classes are the same.

```
∃ hsr: hsRelationship; tm, hm: thMethod |
       nameOfClass(sourceOfHotSpot(hsr)) = nameOfClass(classOfMethod (tm)) ∧
       nameOfClass(targetOfHotSpot(hsr)) = nameOfClass(classOfMethod (hm)) ∧
       (designPattern(hsr) = SEPARATION ∨ (designPattern(hsr) = RECURSIVE_SEPARATION •
              ∃ pbc: pbClass; pbm1, pbm2: pbMethod |
                     nameOfClass(pbc) = nameOfClass(sourceOfHotSpot(hsr)) ∧
                     nameOfClass(pbc) = nameOfClass(targetOfHotSpot(hsr)) ∧
```

```
                    classOfMethod(pbm1)= pbc  ∧
                    classOfMethod(pbm2)= pbc
```

But schemas Separation and RecursiveSeparation define that

```
∀ hsr: hsRelationship; tm, hm: thMethod |
        nameOfClass(sourceOfHotSpot(hsr)) = nameOfClass(classOfMethod (tm)) ∧
        nameOfClass(targetOfHotSpot(hsr)) = nameOfClass(classOfMethod (hm)) ∧
        (designPattern(hsr) = SEPARATION ∨ (designPattern(hsr) = RECURSIVE_SEPARATION •
                (∃ pbc1, pbc2: pbClass; pbm1, pbm2: pbMethod |
                        nameOfClass(pbc1) = nameOfClass(sourceOfHotSpot(hsr)) ∧
                        nameOfClass(pbc2) = nameOfClass(targetOfHotSpot(hsr)) ∧
                        classOfMethod(pbm1)= pbc1 ∧
                        classOfMethod(pbm2)= pbc2)) ∨
                (∃ pbc1, pbc2: pbClass; pbm1, pbm2: pbMethod |
                        nameOfClass(pbc1) = nameOfClass(sourceOfHotSpot(hsr)) ∧
                        nameOfClass(pbc2) = concat(nameOfMethod(hm), "Class") ∧
                        classOfMethod(pbm1)= pbc1 ∧
                        classOfMethod(pbm2)= pbc2))
```

So we have that

```
        (classOfMethod(pbm1)= pbc = classOfMethod(pbm2)) ∧
        (classOfMethod(pbm1)= pbc1 ≠ pbc2 = classOfMethod(pbm2))
```

Which is a contradiction and thus the design loose coupling is proved to be assured by the separation meta-patterns.

Proof of the Design Complexity Property

The unification meta-patterns lead to a simpler design, where the template and hot-spot classes are the same. To prove this property let us suppose that a hot-spot relationship is implemented through a unification meta-pattern and that the template and hook classes are disjoint.

```
∃ hsr: hsRelationship; tm, hm: thMethod |
        nameOfClass(sourceOfHotSpot(hsr)) = nameOfClass(classOfMethod (tm)) ∧
        nameOfClass(targetOfHotSpot(hsr)) = nameOfClass(classOfMethod (hm)) ∧
        (designPattern(hsr) = UNIFICATION  ∨ designPattern(hsr) = RECURSIVE_UNIFICATION •
                ∃ pbc1, pbc2: pbClass; pbm1, pbm2: pbMethod |
                        nameOfClass(pbc1) = nameOfClass(sourceOfHotSpot(hsr)) ∧
                        nameOfClass(pbc2) = nameOfClass(targetOfHotSpot(hsr)) ∧
                        classOfMethod(pbm1)= pbc1 ∧
                        classOfMethod(pbm2)= pbc2
```

But schemas Unification and RecursiveUnification define that

```
∃ hsr: hsRelationship; tm, hm: thMethod |
        nameOfClass(sourceOfHotSpot(hsr)) = nameOfClass(classOfMethod (tm)) ∧
        nameOfClass(targetOfHotSpot(hsr)) = nameOfClass(classOfMethod (hm)) ∧
        (designPattern(hsr) = UNIFICATION  ∨ designPattern(hsr) = RECURSIVE_UNIFICATION •
                ∃ pbc: pbClass; pbm1, pbm2: pbMethod |
                        nameOfClass(pbc) = nameOfClass(sourceOfHotSpot(hsr)) ∧
                        classOfMethod(pbm1)= pbc ∧
```

```
classOfMethod(pbm2)= pbc
```

So we have that

```
(classOfMethod(pbm1)= pbc1 ≠ pbc2 = classOfMethod(pbm2)) ∧
(classOfMethod(pbm1)= pbc = classOfMethod(pbm2)) ∧
```

Which is a contradiction and thus the design complexity properties is proved to be assured by the unification meta-patterns.

8. CONCLUSIONS AND FUTURE WORK

In this paper a viewpoint-based framework design method has been presented in a formal way and its most important properties have been highlighted. Another interesting work in formalizing and proving properties about objects using the Z language can be found in [34], where the information hiding property is described for the COM architectural standard.

This paper shows how viewpoints and the hot-spot relationship can be used as the main design driving force to reusable framework construction. The method provides mechanisms that help the framework developer with the identification of the kernel structure and the flexible parts. The hot-spot card feature is presented here as an approach that can help us to bridge the gap between frameworks and design patterns [29].

The unification step is responsible for harmonically combining the various viewpoints. The use of the hot-spot relationship in this step helps the definition of the framework flexibility requirements. Other approaches to unification of viewpoints can be found in [6].

The proof of the correction of the hot-spot instantiation process using category theory [11] and object calculus [8] is another point of interest that is now being investigated. Some examples of formalization of design patterns using this formalism can be found in [18].

This paper also presents an approach to integrate frameworks with domain specific languages (DSL). We argue that DSLs allows the domain expert to formalize the specification of a software solution immediately without worrying about implementation decisions and the framework complexity. To implement this approach a transformational system (Draco-PUC) is used to transform DSLs specifications into framework instantiation code. It is important to note that DSLs could be transformed into other DSLs, thus creating a domain network, in a way similar with described in [22], providing an easy implementation path for new DSLs.

The method seems to be useful in situations where the frameworks are very complex to build and difficult to use. It will be tested in important and innovative domains, in which we also expect to contribute with the design and development of real frameworks. Two domains that we plan to study soon are electronic commerce and computational biology.

We are now working in the derivation of domain specific languages from problem theory [35], which seems to be an interesting approach not yet been exploited in literature.

Since the Web-based Education (WBE) domain is still not completely understood the need for a framework that supports fast development of alternative WBE environments by non-programmers is a desirable goal. The ALADIN framework presented in this paper as a method case study seems to be an environment where teachers and education researchers can develop their own environments, with little help from software engineers.

A new version of the AulaNet™ environment (http://www.les.inf.puc-rio.br/aulanet) [21] is now being developed with the ALADIN framework. This experiment has two main purposes: the development of a more flexible version of AulaNet™ and further validation of the ALADIN framework.

9. REFERENCES

1. M. Ainsworth, A. H. Cruickshank, L. J. Groves, and P. J. L. Wallis, “Viewpoint specification and Z”, Information and Software Technology, 36(1), 43-51, 1994.

2. P. Alencar, D. Cowan, S. Crespo, M. F. Fontoura, and C. J. Lucena, “Using Viewpoints to Derive a Conceptual Model for Web-Based Education Environments”, MCC17/98, Monografias em Ciência da Computação, Departamento de Informática, PUC-Rio, 1998 (also submitted to Journal of Systems and Software, http://www.les.inf.puc-rio.br/~mafe).

3. J. Bell et al, “Software design for reliability and reuse: A proof-of-concept demonstration”, TRI-Ada’94 Proceedings, 396-404, ACM, 1994.

4. E. Casais, “An incremental class reorganization approach”, ECOOP’92 Proceedings, volume 615 of Lecture Notes in Computer Science, 114-132, 1992.

5. J. Cordy and I. Carmichael, “The TXL Programming Language Syntax and Informal Semantics”, Technical Report, Queen’s University at Kinkston, Canada, 1993.

6. J. Derrick, H. Bowman, and M. Steen, “Viewpoints and Objects”, Technical Report, Computing Laboratory, University of Kent, Canterbury, UK, 1997.

7. E. W. Dijkstra, “The humble programmer”, Communications of the ACM, 15(10), 1972.

8. J. Fiadeiro and T. Maibaum, “Sometimes ‘Tomorrow’ is ‘Sometime’”, Temporal Logic, volume 827 of Lecture Notes in Artificial Intelligence, 48-66, Springer-Verlag, 1994.

9. A. Filkelstein, J. Kramer, B. Nuseibeh, L. Filkelstein, and M. Goedicke, “Viewpoints: A Framework for Integrating Multiple Perspectives in System Development”, International Journal of Software Engineering and Knowledge Engineering,2, 1, 31-58, 1993.

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

11. R. Goldblatt, “Topoi – The Categorial Analysis of Logic”, North-Holland Publishing Company, 1979.

12. P. Hudak, “Building Domain-Specific Embedded Languages”, Computing Surveys, 28A(4), ACM, 1996.

13. R. Iersalimschy, R. Borges, and A. M. Hester, “CGILua A Multi-Paradigmatic Tool for Creating Dynamic WWW Pages”, SBES’97 (Simpósio Brasileiro de Engenharia de Software), 1997.

14. ITU Recommendation X.901-904 – ISO/IEC 10746 1-4, “Open Distributed Processing – Referring Model Parts 1-4”, July 1995.

15. R. Johnson, “Frameworks = (Components + Patterns)”, Communications of the ACM, 40, 10, 1997.

16. R. Johnson and W. F. Opdyke, “Refactoring and aggregation”, In Object Technologies for Advanced Software, First JSSST International Symposium, volume 742 of Lecture Notes in Computer Science, 264-278, Springer-Verlag, 1993.

17. G. Kiczales, J. des Rivieres, and D. G. Bobrow, “The Art of Mataobject Protocol”, MIT Press, 1991.

18. K. Lano, J. C. Bicarregui, S. Goldsack, “Formalizing Design Patterns”, Technical Report, Dept. of Computing, Imperial College, London, UK, 1997.

19. J. C. S. P. Leite, M. Sant’anna, and F. G. Freitas, “Draco-PUC: a Technology Assembly for Domain Oriented Software Development”; Proceedings of the 3rd IEEE International Conference of Software Reuse, 1994.

20. C. Levy, D. Cowan, C. J. Lucena, M. Gattass, and L. H. Figueiredo, “IUP/LED: A Portable User Interface Tool”, Software Practice and Experience (accepted for publication).

21. C. Lucena, H. Fuks, R. Milidiu, L. Macedo, N. Santos, C. Laufer, M. Ribeiro, M. Fontoura, R. Noya, S. Crespo, V. Torres, L. Daflon, and L. Lukowiecki, “AulaNet™ - An Environment for the Development and Maintenance of Courses on the Web”, International Conference on Engineering Education, Rio de Janeiro, Brazil, 1998.

22. J. M. Neighbors, “The Draco Approach to Constructing Software from Reusable Components”, IEEE Transactions on Software Engineering, 10, 5, 1984.

23. W. Pree, “Design Patterns for Object-Oriented Software Development”, Addison-Wesley, 1995.

24. W. Pree, “Framework Patterns”, Sigs Management Briefings, 1996.

25. D. Roberts and R. Johnson, “Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks” in “Pattern Languages of Program Design 3”, Addison-Wesley, 1997.

26. J. Rumbaugh, “Relational Database Design Using an Object-Oriented Methodology”, Communications of the ACM, 31, 4, 1988.

27. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, “Object-Oriented Modeling and Design”, Prentice Hall, Englewood Clifs, NJ, 1991.

28. H. A. Schmid, “Systematic Framework Design by Generalization”, Communications of the ACM, 40, 10, 1997.

29. D. Schmidt, M. Fayad, and R. Johnson, “Software Patterns”, Communications of the ACM, 39, 10, 1996.

30. N. Shu, “Visual Programming”, Van Nostrand Reinhold, New York, 1988

31. C. Simonyi, “The Death Of Computer Languages, The Birth of Intentional Programming”, Technical Report MSR-TR-95-52, 1995.

32. J. M. Spivey, “Understanding Z: A Specification Language and Formal Semantics”, Cambridge University Press, 1988.

33. J. M. Spivey, “The Z Notation: A Reference Manual”, Prentice Hall, Hemel Hempstead, 1989.

34. K. J. Sullivan, J. Socha, M. Marchukov, “Using Formal Methods to Reason about Architectural Standards”, ICSE’97, 503-513, Boston, 1997.

35. W. Turski and T. S. E Maibaum, “The Specification of Computer Programs”, Addison-Wesley Publishing Company, 1987.

36. R. Wirfs-Brock and R. Johnson, “Surveying current research in object-oriented design”, Communications of the ACM, 33, 9, 1990.

37. X. Zhang, “A Rigorous Approach to Comparison of Representational Properties of Object-Oriented Analysis and Design Methods”, Ph. D. Dissertation, Department of Computing and Information Science, Queen’s University, Ontario, Canada, 1997.