A Process for Framework Construction Based on a Pattern Language

Rosana T. V. Braga* and Paulo Cesar Masiero†
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
CP 668 - 13560-970, São Carlos-SP, Brazil
Phone: 55 16 273-9696 Fax: 55 16 271-2238
{rtvb, masiero}@icmc.sc.usp.br

Abstract

A process for building an object-oriented framework based on a pattern language, both for the same domain, is proposed in this paper. This process includes several activities, as the identification of the framework hot-spots, the framework design, implementation, and validation. All these activities are supported by the pattern language. Besides that, using the process proposed, the resulting framework is implicitly documented by the pattern language. This makes it easier for the developer to instantiate applications, as the knowledge about the pattern language is used during the instantiation process.

Keywords: Software reuse, frameworks, software patterns, pattern languages.

1 Introduction

Software reuse can be achieved by several means, among which are class libraries, software patterns and object-oriented software frameworks¹. In particular, frameworks allow the reuse of large software structures in a particular domain, which can be customized to specific applications. Families of similar but non-identical applications can be derived from a single framework. However, frameworks are often very complex to build, understand and use. There are a few methods for framework development, of which the most known are the hot-spot driven framework development [13, 14] and the framework design by systematic generalization [16, 17].

Besides the complexity to build a framework, the effort to learn it is also a problem. According to Fayad et al [9], the time to learn a framework in order to begin to use it can vary

*Financial support from Fapesp Process n. 98/13588-4.
†Financial support from Fapesp/CNPQ.
¹From now on we will use the term framework with the same meaning of object-oriented software framework.
from one to one hundred days, depending on the framework size and comprehensiveness. This factor definitely impacts the final cost of the application and, thus, can inhibit the use of this technology.

Software patterns and pattern languages allow reuse in higher abstraction levels. Software patterns try to capture the experience acquired during software development and to synthesize it in a problem/solution form [1, 10]. A pattern language is a structured collection of patterns that represent the temporal sequence of decisions that lead to the complete design of an application, becoming a method to guide the development process [6, 7].

In this work we show that domain specific pattern languages can be used in the development of a framework for the same domain. The paper is organized as follows. Section 2 presents the related work concerning framework construction and instantiation. Section 3 situates the proposed process in the context of a broader project conducted by the authors. Section 4 presents the framework construction process. Section 5 summarizes the conclusion.

2 Related Work

Pree proposes a process for framework construction [14], in which an object model for a specific application is initially defined and then followed by a construction cycle that is repeated through successive framework refinements. Several steps are included in this cycle. First, the hot spots are identified and documented by hot spot cards. Then the hot spots are designed and implemented, and the framework is tested to assess whether it satisfies the domain requirements. New hot spots may be found in this step and the cycle is repeated.

Schmid states that frameworks are constructed by systematic generalization based on the class model of a fixed application [16, 17]. A high-level hot spot analysis is initially done, aiming at establishing the main aspects of the system that need to be kept flexible. Then each hot spot is analyzed in detail producing its specification. The next step is to make the high level design of each hot spot, generating several hot spot subsystems. The final step is to transform the fixed application class model into the framework class model, replacing groups of classes of the original model by the corresponding hot spot subsystems.

Roberts and Johnson present a pattern language for developing object-oriented frameworks, called “Evolving Frameworks” [15]. This pattern language defines a process for framework construction. It suggests that three concrete applications be developed first and be gradually generalized to produce a white box framework. Then the framework goes through several iterations to become more and more black box, making use of component libraries, pluggable objects, fine-grained objects, and finally of a visual builder and language tools.

Bosch and others present a simple model for framework development with six phases [11]. The first phase deals with the domain analysis, which is necessary to describe the domain covered by the framework and to capture its main requirements and concepts, resulting in a domain analysis model. In the second phase the framework architectural design is created. In the third phase this design is refined. In the fourth phase the framework is implemented using a specific programming language. In the fifth phase the framework is tested to evaluate both
its functionality and usability. Finally, in the sixth phase the framework is documented, usually with a user guide and design documentation.

The processes suggested by Pree, Schmid, and Roberts and Johnson begin with particular application models, including the desired flexibility later, while Bosch’s approach is to obtain the domain analysis model at the beginning, which makes the framework hot spots more foreseeable. Our work follows Bosch’s rationale, as we use a pattern language to capture the application domain and to guide the framework development, particularly to identify its hot spots at the beginning, to minimize iteration cycles and, at the same time, to ease the subsequent framework learning and usage. Our process is similar to the other processes mentioned above with respect to the design phase, but it is guided by the pattern language.

3 Contextualization

We propose a process for framework construction based on a pattern language for the same domain. This process is part of our approach for software reuse, illustrated in Figure 1. This approach has four steps, of which the process proposed in this paper takes care of the second step. The other steps are not treated in detail, but are briefly summarized.

Figure 1: Approach for Framework Construction and Instantiation

The first step consists of developing a pattern language for a specific domain. The experience
acquired during software development in a specific domain or reverse engineering of existing systems can be used to accomplish this. It involves domain analysis, splitting the problems found in the domain into smaller problems, and creating a pattern to solve each of these problems. In the second step, the pattern language is used to develop a white-box framework for this domain. The main goal of this framework is to allow code reuse for all classes belonging to the pattern language. The process to be followed in order to achieve this is explored in section 4. In the third step a wizard to help instantiating the framework is developed, based on the pattern language and on the white-box framework. This step is optional, as it has a high cost that must be balanced against the long term gains in ease of use. Finally, in the fourth step, the framework can be instantiated by using its white-box version (a) or by using its wizard (b). In both cases it is necessary to supply the requirements of the specific application to be built, and the result is the corresponding specific system.

The proposed approach is illustrated using the GREN framework [2], constructed based on the Pattern Language for Business Resource Management (GRN) [4]. The GREN wizard [3] is an example of a tool that helps instantiating the GREN framework.

The GRN pattern language was built based on the experience acquired during development of systems for business resource management. Business resources are assets or services managed by specific applications, as for example videotapes, products or physician time. Business resource management applications include those for rental, trade or maintenance of assets or services. The GRN pattern language has fifteen patterns that guide the developer during the analysis of systems for this domain. The first three patterns concern the identification, quantification and storage of the business resource. The next seven patterns deal with several types of management that can be done with business resources, as for example, rental, reservation, trade, quotation, and maintenance. The last five patterns treat details that are common to all the seven types of transactions, as for example payment and commissions. Figure 2 shows part of the **Quantify the Resource** pattern, extracted from the extended version of the GRN pattern language. It can be observed that this pattern has four alternative solutions, depending on how the business resource is quantified. This will derive a framework hot-spot in the step that follows.

4 Framework Construction

The overall process for framework construction based on a pattern language is illustrated in Figure 3. This process has four steps, detailed in the following sections. Basically, it consists of taking the pattern language as input and producing the framework as output.

4.1 Hot Spots Identification

In step 2.1 the framework hot spots are identified. This is basically done using information present in the pattern language. This activity has five sub-steps, explained in detail elsewhere [5], and summarized as follows. The first sub-step analyzes the pattern language graph, if there is one, or the “Following Patterns” section. The main goal is to find hot spots implied by optional patterns, because some domain functionality associated to a pattern may be optional.
Pattern 2: QUANTIFY THE RESOURCE

Context
You have identified a resource that your application deals with and its relevant qualities. An important issue to be considered now is the form of resource quantification. There are certain applications in which it is important to trace specific instances of a resource, because they are transacted individually. For example, a book in a library can have several copies, each lent to a different reader. Some applications deal with a certain quantity of the resource or with resource lots. In these applications, it is not necessary to know what particular instance of the resource was actually transacted. For example, a certain weight of steel is sold. In other applications, the resource is dealt with as a whole, as for example a car that goes to maintenance or a doctor that examines a patient.

Problem
How does the application quantify the business resource?

Forces
- Knowing exactly what is the form of quantification adopted by the application is important during analysis. A wrong decision at this point may compromise future evolution.
- If tracing specific instances of a resource is needed, then redundant information could be stored for the several instances of the same resource, but this redundancy would be undesirable.
- . . .

Structure
There are four slightly different solutions for this problem, depending on the form of quantification. Figures 4 through 7 show the four QUANTIFY THE RESOURCE sub-patterns.

When it is important to distinguish among resource instances, use INSTANTIABLE RESOURCE sub-pattern (Figure 4). When the resource is managed in a certain quantity, use the MEASURABLE RESOURCE sub-pattern (Figure 5). When the resource is unique, use the SINGLE RESOURCE sub-pattern (Figure 6). When the resource is dealt with in lots, use the LOTABLE RESOURCE sub-pattern (Figure 7).

Participants
. . .

Example
. . .

Forcing patterns
After you Quantify the Resource, examine your application to verify whether it is important to know about the resources storage. If so, try to apply STORE THE RESOURCE (3). If not, proceed examining your application to verify which kind of resource transactions are done. If the application concerns resource location or rental, you should apply RENT THE RESOURCE (4). If the application concerns resource trading, i.e., resource purchase or sale, you should apply TRADE THE RESOURCE (6). If the application deals with resource repair, you should apply MAINTAIN THE RESOURCE (9).
for certain applications. So, look for paths that skip one or more patterns, as they will indicate possible framework hot spots. In the second sub-step, analyze each pattern of the pattern language, looking at its constituent sections, which can indicate several framework hot spots. In the “variants” or “sub-patterns” section you can find alternative solutions for the problem solved by the pattern and variable aspects that must be available for the framework user to instantiate it. For example, in Figure 2 there are several alternative solutions for the problem, and they are shown in different sub-patterns of the solution, like “InstantiableResource” and “MeasurableResource”. The “participants” and “collaborations” sections, which are present in patterns that follow the GoF format [15], can indicate some of the framework hot spots, e.g., an optional participant is an indication of a hot spot. The “implementation” section contains suggestions of alternative implementations of the proposed solution, so that according to the restrictions imposed by each particular application, different implementations can be chosen. Thus, they can indicate another type of framework hot spot. Another source of hot spots is the “structure” section, which contains a diagrammatic representation of the pattern classes and their relationships. A detailed analysis of this section can help identifying alternative behaviors, often not described in the “participants” section. Thus, new hot spots can be defined to allow, for example, new attributes or methods for the classes and alternative algorithms for computing attributes.

In the third sub-step, refine the specification of each hot spot to include enough information for its subsequent design and implementation. Also, identify other hot spots that are not explicit in the pattern language, but should be included because they would bring more flexibility to the framework. This can be done based on your general knowledge about hot spots implementation. You should use the information about these new hot spots as feedback to improve the pattern language. In the fourth sub-step, look at the list of framework hot spots produced until now and make a cross-reference among them. This can help you to identify non-consistency in the whole list, which might imply in a few more hot spots. You need to use your domain knowledge to perceive this. Here, like in the third step, the new hot spots discovered are used as feedback.
to improve the pattern language. In the fifth sub-step, consider other non-functional aspects of the application that might originate new hot spots, which include portability, usability, security and reliability. Also consider design and implementation issues that would bring more flexibility to the framework.

The result of step 2.1 is a list of the framework hot spots, composed of an identification code, a description of the hot spot, the hot spot type, the associated source in the pattern language (optional) and the pattern where this hot spot was identified. The hot spot type contains information about what has to be done to obtain an application from the framework. For example, some hot spot types are:

- **PARTIC_CHOICE** establishes that there will be a choice among participants of a pattern, i.e., the framework user will have to choose which participant will be used in a particular instantiation;

- **PATTERN_OPTION** determines that the whole pattern may be optionally applied in specific applications;

- **PARTIC_OPTION** allows a participant of a pattern not to be used in particular instantiations;

As an example, Table 1 shows three of the thirty-five hot spots found in this step for the GREN framework [5]. For example, hot spot number 2 was derived from pattern number 2 - Quantify the Resource (see Figure 2) by analyzing its participants, structure and variants (sub-patterns). This hot spot allows four different types of quantification for the resource, according to the four sub-patterns presented in the pattern structure: Single Resource, Measurable Resource, Instantiable Resource and Lotable Resource. It is a **PARTIC_CHOICE** pattern, because the framework user will choose the participant classes according to the quantification strategy chosen. Notice that the hot spots identified in the GREN framework, using the GRN pattern language, are situated at the analysis level, because the fifth sub-step was not applied in this example.

<table>
<thead>
<tr>
<th>Hot spot #</th>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Source in the pattern language</th>
<th>Pattern #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Resource Qualification</td>
<td>A resource can have a type, but this is optional. It can also have multiple types or nested types.</td>
<td>PARTIC_CHOICE</td>
<td>Participants, Variants</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Resource Quantification</td>
<td>A resource can be unique, can have multiple instances, can be managed in quantities or in lots.</td>
<td>PARTIC_CHOICE</td>
<td>Participants, Structure, Variants (sub-patterns)</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Resource Storage</td>
<td>It can or cannot be desirable that the application manages the resource storage.</td>
<td>PATTERN_OPTION</td>
<td>Language Graph + Context</td>
<td>3</td>
</tr>
</tbody>
</table>
4.2 Framework design

Based on the resulting list of framework hot spots and on the pattern language, the framework is designed (step 2.2). We assume that the patterns of the pattern language contain a section with the structure of each pattern. This structure is a small analysis model for the pattern and can be used to design the framework part that takes care of the problem solved by the pattern.

Two sub-steps are involved in the framework design. The first sub-step takes care of the framework architectural design. Here, the developer needs to choose the software architecture that best fits the requirements for the framework. This includes making decisions about the persistence mechanism, the graphical user interface, security issues, and distribution.

As an example, the GREN framework architecture was designed using three layers: a persistence layer, an application layer and a graphical user interface (GUI) layer. The persistence layer isolates the operations that deal with object persistence, i.e., it introduces a layer through which the other layers can invoke methods without worrying about how the persistence will be done. The application layer was created according to the GRN patterns, so that the classes and relationships contained in each pattern have the corresponding implementation in this layer. The GUI layer implements the classes that allow the final user to interact with the system.

The second sub-step consists of creating the class hierarchy for the framework, based on the structure proposed by each pattern of the pattern language, on the framework hot spots list, and on design patterns and metapatterns. This involves several activities. Initially, an overall framework diagram is created that combines all the classes contained in each pattern of the pattern language, including sub-patterns and variants. This overall diagram is then refined through object-oriented mechanisms, i.e., generalization and specialization. The resulting diagram is then analyzed to investigate the possibility of using design patterns [10] or meta-patterns [13] to achieve the desired flexibility. This investigation involves not only making some design decisions, but also having knowledge about usual techniques used in framework implementation. The resulting diagram represents, basically, the functionality provided by the pattern language. Then, the framework architectural model (produced in the first sub-step) is analyzed, and new classes are created to represent other elements of the framework architecture, presumably not included in the pattern language. The result of this phase is the framework design model.

As an example, Figure 4 shows part of the GREN framework design model obtained using this approach, in particular the classes related to “Resource” are shown. Classes “Static Object” and “Qualifiable Object” were created during the generalization/specialization, as there are several classes that can inherit their behavior. The Strategy design pattern [10] was used to model hot spot number 2 of the GREN framework, described in Section 4.1. Each resource object refers to a strategy object (“QuantificationStrategy” of Figure 4), which is responsible for its quantification aspects, allowing the framework to implement the four different solutions required by the hot spot. The “Persistent Object” class was created to model the persistence layer.
4.3 Framework implementation

Step 2.3 takes care of implementing the framework, based on the framework design model. Two sub-steps are involved in this task. In the first sub-step, the framework classes defined in the framework design model are implemented, using a particular programming language. The pattern language is used to solve any doubts the developer may have about the functionality of the participating classes, as it is an excellent source of information about the domain. Other specific implementation decisions need to be made here as, for example, data structures to be used, widgets to be used in the graphical user interface, style of end-user reports. The result of this sub-step is the framework code. Notice that this first sub-step may be very complex, depending on the framework software architecture defined in the previous step. Software components or classes may be reused from other projects and tools can be used to ease this step.

The pattern language can be used to guide the framework implementation in an incremental way. Each pattern may be considered as a functional unit, so that the implementation can begin with the first pattern and follow with the other patterns until the whole framework is
done. This is not always easy to do: during the design phase (step 2.2), the participant classes of each pattern may have been decomposed and specialized. So, the framework design model may contain classes that belong to several patterns.

The second sub-step takes care of documenting the framework. Following our guidelines, the documentation produced in this step will identify the mapping between the framework classes obtained in the previous sub-step and the pattern language. The goal is to ease the framework instantiation to specific applications. Tables are produced to map the applied patterns and variants to the corresponding classes of the framework that need to be sub-classed to produce specific applications. Having the history of the patterns and variants applied, these tables are consulted to identify which concrete classes need to be created (the tables will indicate the correct super-classes to use) and which methods need to be overridden. Hence, framework users can easily know what needs to be done in each framework part, according to how the corresponding pattern of the pattern language was used to model their application.

We recommend the creation of two mapping tables in this sub-step: a “classes mapping table” and a “methods mapping table”. The “classes mapping table” establishes the relationship between participant classes of each pattern and framework classes. When instantiating applications using a framework, developers must create new classes that refer to the specific application. These new classes must inherit from the appropriate classes in the framework. In our approach, the instantiation is supported by the pattern language, so the developer knows which patterns, variants or sub-patterns are used, as well as the roles played by each application class in the corresponding pattern. Thus, this table should contain the pattern name, the variant or sub-pattern, the participant classes of the pattern, and the corresponding classes in the framework. This mapping is not necessarily one to one, as a participant class of the pattern may have been implemented in the framework through several classes. So, this table should provide means of letting the developer know which classes to create for each situation.

The “methods mapping table” indicates which methods should be overridden in the newly created classes. In fact, it documents the framework hook methods [13], i.e., methods declared in abstract classes of the framework and which code is expected to be supplied by the new application classes. This table should have three columns: the first with the superclass name, i.e., the name of a framework class that has abstract methods to be overridden by any descendant classes; the second with the method signature (name, parameters and return type); and the third with a complete description of the method, how to override it, and a sample code.

Let us illustrate this step with the GREN framework. Its implementation was done using the VisualWorks Smalltalk [8] and the MySQL database [12] for object persistence. The first version of the GREN framework contains about 150 classes and 26k lines of code in Smalltalk. As mentioned in section 4.2, GREN was implemented using three layers: persistence, application, and GUI. The persistence layer was implemented first, followed by the application and GUI layers, which were implemented in an incremental way, following the patterns of the pattern language. For example, application layer classes belonging to pattern 1 - Identify the Resource were implemented, followed by the corresponding GUI layer classes. After testing these classes by instantiating the little framework obtained, we have proceeded with pattern 2, and so on.

The GREN cookbook [2] contains six tables that allow the mapping between GRN and
GREN. The first two tables compose the “classes mapping table” and the last four tables compose the “methods mapping table”. The “classes mapping table” was split into two tables to separate application classes from GUI classes. The “methods mapping table” was split into four tables to separate mandatory methods to be overridden from the optionally overridden ones, and to separate the application layer methods from those of the GUI layer.

Table 2 shows part of the “classes mapping table” for the GREN framework. Notice that columns 1 (pattern) and 2 (variant) ease the framework usage, because the developer can skip the table rows that refer to patterns that were not applied during instantiation. So, the focus is on the patterns and variants applied. The third column (pattern class) contains the participant classes of the pattern. So, if a specific participant was not used during instantiation, the corresponding row can also be skipped. The fourth column (framework superclass) indicates the framework superclass to be used during instantiation, i.e., the concrete class playing the specific role needs to inherit from the framework class indicated in column four. The fifth column (reference code) is just a cross reference to be used in the second mapping table. The sixth column was added to the “classes mapping table” to inform whether or not it is necessary to create a table in the database to store the objects of such class, as the GREN framework uses a relational database for object persistence.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Variant</th>
<th>Pattern Class</th>
<th>Framework Class</th>
<th>Ref Code</th>
<th>Create Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - Identify the Resource</td>
<td></td>
<td>Resource</td>
<td>Resource</td>
<td>N1</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Default or Multiple</td>
<td>Resource Type</td>
<td>SimpleType</td>
<td>N2</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Nested Types</td>
<td>Resource Type</td>
<td>NestedType</td>
<td>N2</td>
<td>Yes</td>
</tr>
<tr>
<td>2 - Quantify the Resource</td>
<td></td>
<td>Resource Instance</td>
<td>ResourceInstance</td>
<td>N3</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Instantiable Resource</td>
<td>Resource Instance</td>
<td>ResourceInstance</td>
<td>N3</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Measurable or Lotable Resource</td>
<td>Measure Unity</td>
<td>MeasureUnity</td>
<td>N4</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Lotable Resource</td>
<td>Resource Lot</td>
<td>ResourceLot</td>
<td>N5</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The framework that results from this step can be used to implement specific applications. Using the mapping tables produced according to our guidelines, the user does not need a deep knowledge of the framework design details, because the pattern language usage will help the instantiation process.

4.4 Framework validation

Step 2.4 is responsible for validating the framework, using the produced framework, its cookbook, and the requirement documents for several specific applications. The framework will be tested for these particular applications before it can be deployed for general use. So, to better validate it, it is desirable to test it for as many applications as possible. The goal is to check whether the framework is working according to its purposes. The pattern language can help
the framework developer to define which applications should be tested. For example, a set of applications could be created to exercise, at least, all the patterns of the pattern language, including all their variants and sub-patterns.

Observe that framework validation is only possible by instantiating it to concrete applications, i.e., the validation is done using the same instantiation process used after it is deployed. The framework instantiation process (Step 4 of Figure 1) will be shown in detail in a future work, but we summarize it below.

We suggest four sub-steps to validate the framework, although the specific framework documentation should be consulted during the instantiation process. For example, in the GREN framework there is a cookbook that contains all the guidelines to map the requirements into the framework, using the pattern language.

In the first sub-step, for each different application to be tested, the requirements of the specific application are analyzed using the pattern language, producing its analysis model. Then, in the second sub-step, the mapping between the analysis model produced and the framework classes is done, using the “classes mapping table” and the “methods mapping table”. In the third sub-step, the concrete application classes are created, using a specific programming language, which is the same language used in the framework implementation. The result is the specific application code, which needs to be tested (fourth sub-step). The test activity is out of the scope of this work, but it requires the application code compilation, installation of the framework software at a client machine, database creation, and selection of a test strategy. Errors found during this phase need to be fixed. Generally, these errors belong to the framework and not to the specific application, because we are in the framework validation step.

After the framework has been instantiated to several different applications and the errors found have been corrected, the framework can be released for use. However, new errors can be found when it is instantiated to applications that differ from the ones it was tested to. Besides that, the framework user can introduce errors through the adaptations done to the framework.

The GREN framework has been tested for three different applications, carefully chosen in order to exercise all the GRN patterns. It was not possible to cover all variants and sub-patterns, due to the short time we had to test it, but we are in the process of doing it now. The first application was for a car repair shop, the second for a video rental store and the third for a shop that sells and rents products for parties. After theses tests were finished, we have done some case studies with undergraduate students. Their task was to develop two applications, specifically for a hotel and for a car rental shop, based on supplied requirements. Twelve groups were formed and six of them have developed the hotel system, while the other six have implemented the car rental shop. They have used the GRN pattern language to model the system and the GREN framework to implement it in VisualWorks Smalltalk, following the GREN cookbook. Several errors could be found and corrected in the framework and in the cookbook due to these new instantiations.
5 Concluding remarks

The approach here proposed intends to ease framework development and instantiation, through the use of a pattern language. We guide the framework construction in all its phases, beginning with the hot spots identification, design, implementation, and validation. Frameworks built using our approach have its architecture influenced by the pattern language, which eases the instantiation of new applications. With the special documentation suggested by our approach, framework users need only to know about the pattern language usage in order to instantiate the framework. No technical knowledge about the framework class hierarchy or implementation details are necessary. The novel aspect of our proposal is to use the pattern language as an integrator element of the technology.

Framework instantiation can imply in improvements both in the pattern language and in the framework itself. New requirements discovered during instantiation and that are not supported by the framework can be used as feedback to improve the pattern language and to evolve the framework. The framework user needs to decide whether to make the adaptations only in the application being instantiated or to communicate them to the framework developer. The former is the easiest decision to make, while the latter is the one that contributes for the framework evolution.

Not all types of pattern languages could help framework construction and instantiation. The candidate pattern languages are those for domains in which constructing a framework makes sense and would be useful; those that concern problems to be solved during analysis, design and implementation of systems in the domain; and, preferably, those that show design or analysis models. This excludes pattern languages comprised by process, architectural or organizational patterns. Information systems are an example of a domain that is suitable for this approach.

References


[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Massachusetts, 1994.


