

# Finding Frameworks Hot Spots in Pattern Languages

**Rosana T. V. Braga**

**Paulo Cesar Masiero**

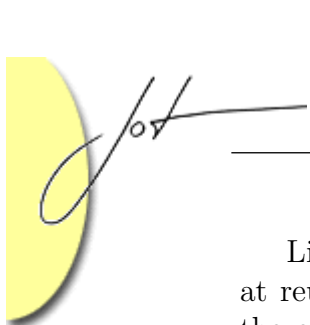
Instituto de Ciências Matemáticas e de Computação  
Universidade de São Paulo — Brazil

An important issue that contributes to the complexity of object-oriented framework development is the identification of its hot spots, i.e., the framework parts that must be kept flexible, as they are specific of individual systems. A process for identification of hot spots in an analysis pattern language is proposed. Several types of hot spots are identifiable from information presented in the elements of each pattern of the pattern language, making possible to define a process to sistematize this task. An example illustrates the hot spots identification based on a pattern language for business resource management.

## 1 INTRODUCTION

Software reuse is a goal that was set almost simultaneously with software engineering. Structured programming, followed by object-oriented programming and domain analysis were achievements obtained a long time ago to enhance software reuse. Object-oriented software frameworks have emerged in the same context. They allow the reuse of large structures in a particular domain, which can be customized to specific applications in the domain. Families of similar but non-identical applications can be derived from a single framework [Johnson and Foote, 1988], [Fayad and Schmidt, 1997].

The difficulty to build, understand, and use frameworks has motivated the research around frameworks, which includes methods for framework development and documentation. In particular, a major complexity in framework development concerns the identification of its hot spots, i.e., the parts that have to be kept flexible, as they are specific of individual systems. This involves deep knowledge about the domain for which the framework is being built, as hot spots are designed to be generic and further adapted according to the requirements of each instantiated application. Hot spots are usually discovered by domain analysis and then by successive framework refinements. However, each new discovery may imply in the need to redesign part of the framework, which makes development more complex. The best approach is to know beforehand which are the framework hot spots, in order to minimize the number of iterations needed for its construction.



Like frameworks, software patterns and pattern languages have emerged aiming at reuse, but in higher abstraction levels. While software patterns try to capture the experience acquired during software development and synthesize it in a problem/solution form [Gamma et al., 1995], a pattern language is a structured collection of patterns that build on each other to transform needs and constraints into an architecture [Coplien, 1998]. Pattern languages represent the temporal sequence of decisions that lead to the complete design of an application, so it becomes a method to guide the development process [Brugali et al., 2000].

Pattern languages and frameworks can be used together to achieve even more reuse. Both are conceived for a specific domain, solving most of the problems that are common to applications in that domain [Brugali and Menga, 1999]. A pattern language can be used for documenting the framework, as already shown in several works [Aarsten et al., 2000, Johnson, 1992]; for supporting the framework design and implementation [Brugali and Menga, 1999, Brugali et al., 2000]; and as a method to guide the transformation of the framework in a concrete application [Brugali and Menga, 1999]. As pattern languages contain the main abstractions found in an application domain, they have built-in information about the points that differ from one application to another. So, they are a valuable source to identify the framework hot-spots. In particular, this work considers analysis pattern languages, which are composed of patterns to solve problems found during system analysis. These patterns are placed in a higher abstraction level than design patterns, as they are domain specific.

The work shown in this paper depends on the existence of a pattern language for a specific domain. Details about the construction of this pattern language is out of the scope of this paper and can be found elsewhere [Braga and Masiero, 2003]. Basically, it involves domain analysis, the use of experience acquired during software development in a specific domain, or the reverse engineering of existing systems. After building a domain model to represent applications in a specific domain, the problems found in the domain are split into smaller problems, and an analysis pattern is created to solve each of these problems. The interaction among the patterns is documented, for example using a graph, and the patterns are written.

The pattern language is not constructed only with the purpose of helping identifying the hot spots of a possible framework. Continuing the work presented in this paper, we have defined two different processes: the first to build frameworks based on pattern languages [Braga and Masiero, 2002b] and the second for using a pattern language during the instantiation of applications with a framework built based in that pattern language [Braga and Masiero, 2002c]. This second process is supported by a wizard with a user interface that follows the same concepts of the pattern language [Braga and Masiero, 2002a], so that users can apply the pattern language and have their systems semi-automatically built.

This work focuses on analysis pattern languages as a source for framework hot spots mining, pointing to the several types of hot spots that can be found inside pattern languages and proposing guidelines of how to find them. The purpose is to ease framework construction by minimizing the iteration cycles needed to refine it.



The paper is organized as follows. Section 2 presents the problem of hot spots elicitation and shows the related work. Section 3 describes the types of hot spots that are identifiable from pattern languages. Section 4 proposes guidelines to identify hot spots in a pattern language. Section 5 establishes the relationship between the hot spot type and the design of the framework. Section 6 presents a case study, in which a pattern language for business resource management is the source for identifying the hot spots of its associated framework. Section 7 presents the concluding remarks.

## 2 HOT SPOTS ELICITATION

A framework is a powerful technique to improve reuse, as lots of different applications can be obtained by instantiating it. However, the instantiation process is usually very complex, requiring a deep understanding of the framework design and implementation. To achieve the desired flexibility, frameworks contain special constructions that difficult its understanding. All frameworks have a fixed part, called frozen spots [Pree, 1999], that reflect the common behavior of applications in the domain. On the other hand, frameworks have parts that need to be kept flexible, called hot spots, which have to be adapted according to the specific requirements of concrete applications derived from the framework.

In most existing techniques for framework development [Pree, 1995, Pree, 1999, Schmid, 1997, Schmid, 1999, Roberts and Johnson, 1998], hot spots are identified throughout the process. They begin with a particular application model, which is used to define the first framework version, and then it is refined through several iteration cycles, including more and more hot spots. In other approaches, like Bosch's [J. Bosch and Fayad, 1999], a domain analysis model is obtained at the beginning, which makes the framework hot spots more foreseeable.

Similar to identifying frameworks hot spots is identifying the variability of a software product line [Griss et al., 1998, Svahnberg et al., 2001]. In this case, the variability is documented by the system *features*, which are used to group related requirements concerning the system behavior. Svahnberg et al. [Svahnberg et al., 2001] classify features in four groups: External features, mandatory features, optional features, and variant features. The approach presented in this paper also classifies the several types of hot spots, some of which are similar to the features classification.

If an analysis pattern language exists for a particular domain, then it can be used as a source for hot spots elicitation, minimizing iteration cycles needed for framework construction, like in Bosch's approach. Additionally, the framework can be designed based on the pattern language, easing its future instantiation, as the requirements of the the specific application will be more easily mapped to hot spots that have to be adapted [Braga and Masiero, 2002b, Braga and Masiero, 2002c].

### 3 TYPES OF HOT SPOTS IDENTIFIABLE FROM PATTERN LANGUAGES

There are several types of hot spots in a framework that must be adapted to produce specific applications. Usually, this adaptation involves adding new classes and overriding methods, with the purpose of characterizing the specific application behavior. For example, in white box frameworks reuse is obtained through inheritance, i.e., the concrete application classes inherit from framework abstract classes, whose hook methods have to be overridden to provide the desired behavior.

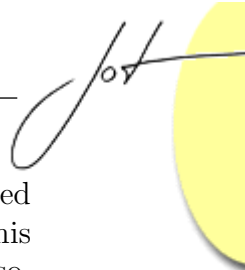
The existence of a pattern language for a particular domain can greatly help in the hot spots identification. Table 1 summarizes the types of hot spots that can be identified from a pattern language. A code is assigned to each of them to ease future reference. The adaptation type offers insights of how to design and implement it in the framework and of how do adapt it during instantiation. The following subsections describe how to find these types of hot spots in a pattern language.

Table 1: Types of Hot spots identifiable from pattern languages

IdCode	Hot spot Description	Adaptation type	Main sources in the pattern language
PATTERN_OPTION	Optional pattern	Several classes and relationships are disabled	Language graph, Following patterns, Related patterns
PARTIC_OPTION	Optional participant	One class and its relationships with other classes are disabled or enabled	Participants
PARTIC_CHOICE	Choice of participants	One or more participants must be chosen according to the system requirements	Participants, Structure, Variants
RELATIONSHIP	Change of Relationship	One or more relationships must be changed according to the system requirements	Participants
BEHAVIOR	Change of Behavior	One or more algorithms must be changed according to the system requirements	Participants, Structure
PROPAGATION	Propagation effect as a consequence of other pattern application	Some participants may have changes in attributes or methods according to other patterns already applied	Participants, Structure

#### Optional patterns

A hot spot belongs to the type PATTERN\_OPTION if the whole pattern can be considered optional when instantiating applications. It can be found by analyzing the general structure of a pattern language, i.e., it is necessary to observe the several inter-related patterns. The relationship among patterns is generally shown by a graph that represents, rather than the interaction among patterns, the sequence in which they are applied. A simple analysis of which patterns are of mandatory use and which are of optional use, indicates several framework hot spots. For example, both the pattern language for improving the capacity of reactive systems



[Meszaros, 1995] and the pattern language for business resource management, named GRN [Braga et al., 1999] use a graph to illustrate their structure. By analyzing this graph we can have an idea of which patterns are mandatory or optional and, so, we can identify several hot spots of the framework to be built. For example, in the GRN pattern language, the RESERVE THE RESOURCE pattern is optional. This indicates the need of a hot spot in the framework to handle this feature. This can be confirmed by analyzing the pattern context which presents the scenario for the pattern usage.

If the pattern language has no corresponding graph, the information necessary to know whether each pattern is optional or not, can be found mainly in sections “Context”, “Following patterns”, and “Related patterns”. The context is important, as it gives indications of the desirable features for the pattern usage. Adding to this the knowledge about the application domain, we can identify applications that do not fit in the context and that, consequently, do not use the pattern. Sections “Related patterns” and “Following patterns” show other patterns related to the current pattern, helping to identify other alternative patterns and, consequently, indicate the pattern optionality. The pattern language Accounts and Transactions [Johnson, 1996] — although written in the Alexander form and, consequently, without a “related patterns” section — has a paragraph at the end of the “solution” section pointing to the related patterns, where it is clear that the user is directed to alternative patterns according to the application characteristics.

## Optional participants

The PARTIC\_OPTION hot spot type denotes that a pattern participant is optional when instantiating concrete applications, i.e., the pattern can be applied without one of its participants. This type of hot spot can be found at different sections of a pattern language. The main sources are the “participants” and “collaborations” sections, which are present in patterns that follow the GoF format [Gamma et al., 1995]. As they describe the participants of the pattern and their collaboration, they provide alternatives of using or not some of the participants. When a participant is optional, there is a description of how the pattern works without it. For example, in the GRN pattern language, “Source-Party” is an optional participant in the RENT THE RESOURCE pattern, because small organizations do not have branches or departments to be managed.

The “variants” or “variations” section, commonly found in the patterns of a pattern language, is one of the richest sources of hot spots, mainly of the PARTIC\_OPTION type, because the variants often present solutions that differ with respect to the participants.

## Choice among participants

When a pattern participant does not specify a particular class, but gives alternatives of two or more classes that can be used, a decision has to be made during analysis to define the participant class, according to the requirements. This type of hot spot (PARTIC\_CHOICE) can be detected in the “Structure” section of the pattern language, where a class diagram presents the solution, or in the “Participants” section, where the classes that make the pattern are described. For example, in the pattern language for Object-RDBMS Integration [Brown and Whitenack, 1996] there is a section named “Discussion” in which variations of the solution are presented to discuss the participants and their collaborations.

## Relationship and Behavior

The RELATIONSHIP and BEHAVIOR hot spot types can be found in several sections of the pattern language. The RELATIONSHIP type consists of changing a relationship between classes to obtain the desired functionality. The BEHAVIOR type consists of changing a method (or operation) to attend a specific requirement. The “Implementation” section, present in patterns that follow the GoF format, 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, this section is a good source of these types of hot spot. It must be observed that the framework developer often makes implementation choices that limit the possible implementations to one or two solutions. So, it is common for the framework not to cover all the possibilities presented in the “Implementation” section.

Another source of these types 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 to identify alternative behaviors that may be desired for the system operations, 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.

It is possible to misunderstand the hot spot types BEHAVIOR and RELATIONSHIP. In fact, the RELATIONSHIP type is more specific and can include the BEHAVIOR type, because when a class relationship is changed there is a change of behavior in the system. In this case, it is better to use the more specific type. An example of the BEHAVIOR hot spot type is the payment of commissions regarding transactions performed by the executor in the “Identify the Transaction Executor” pattern of the GRN pattern language. The “variants” section of this pattern suggests two different ways of implementing the commission payment: the payment can be done independently of the corresponding payment by the customer, or it can be done provided that the payment of each installment is done by the customer. So, the classes and relationships are the same, but the algorithm used to compute commissions can be different.



## Propagation

The PROPAGATION hot spots are not very easy to find, because they reflect decisions made in previously applied patterns. This means that some patterns may have their participants or relationships modified according to the pattern variants already applied. For example, in the GRN pattern language, the choice done when applying the QUANTIFY THE RESOURCE pattern implies in several additions to the participants of patterns applied subsequently.

Like the BEHAVIOR and RELATIONSHIP types, the PROPAGATION type is also prone to be misunderstood. It is possible to categorize this type of hot spot into one of the five other types, but it is recommended to classify it as PROPAGATION to ease its further design: there is a high probability that its design will follow the design of the hot spots for which it is a propagation of.

## 4 GUIDELINES FOR IDENTIFYING HOT SPOTS

Based on the types of hot spots defined in Table 1 and on the information contained in a pattern language, a generic process is proposed to help the framework developer to identify the hot spots using a pattern language. Figure 1 shows the steps involved in this process.

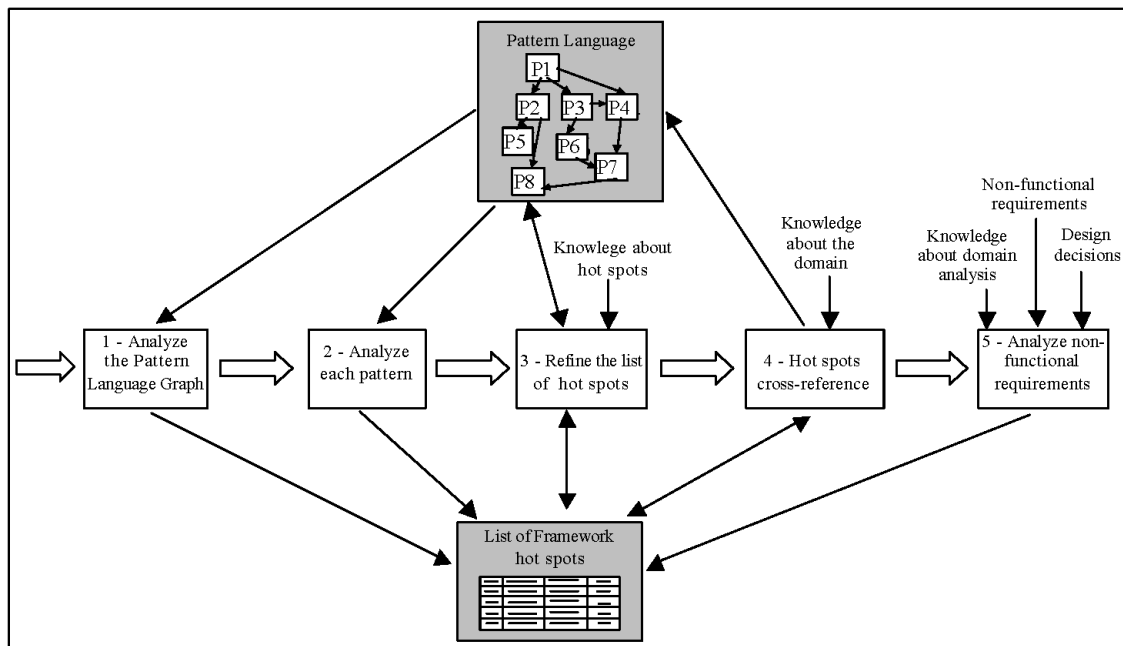
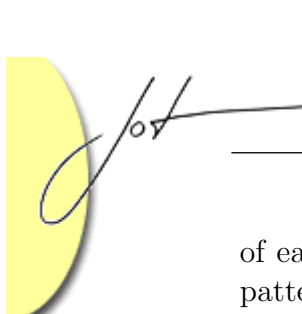


Figure 1: Process for hot spots identification based on a pattern language

In the first step, an analysis is done in the pattern language graph if there is one. Alternatively, the sections “Following Patterns”, “Related Patterns”, or “Context”



of each pattern are studied. The main goal is to find paths that skip one or more patterns, so that the optional patterns are determined. As explained in Section 3, if a pattern can be optionally applied during the pattern language usage, then the corresponding framework needs to have means of working correctly without this pattern. So, one or more hot spots are included in the framework to cope with this behavior. In this case, the hot spot type is `PATTERN_OPTION` and the framework user can enable or disable it, resulting in the inclusion or omission of one or more classes that compose this pattern.

In the second step, each pattern of the pattern language is analyzed, by studying its constituent sections, as they can indicate several framework hot spots. The explanations supplied in sections 3 to 3 are useful to identify hot spots in this step.

In the third step, a refinement is done to each hot spot, aiming at supplying the information necessary for its further design and implementation. It is common to discover new hot spots in this step, depending on the experience of the framework designer, because to refine a hot spot it may be necessary to split it into two or more hot spots. The information about new hot spots can be used as feedback to improve the pattern language in the subsequent iteration cycle.

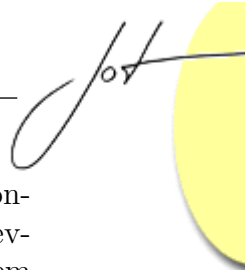
To better organize the hot spots list, it is recommended to build a table with information about hot spots. For each hot spot identified in steps 1 to 3 do the following:

1. Include it in the hot spots table, assigning it a number, a name and a brief description of the desired flexibility.
2. Inform the hot spot type, according to what is needed to adapt the framework for a specific application (Table 1).
3. Associate the hot spot with its source in the pattern language and the pattern number.

In the fourth step, a cross reference among the hot spots is done to identify possible inconsistencies in the whole list. New hot spots can be found in this step, for example due to the incorrect propagation of previously applied patterns. Domain knowledge is essential to perform this activity and, like in the previous step, new hot spots can be used to enhance the pattern language.

In the last step, other non-functional aspects of the application with potential to originate new hot spots are considered, which include portability, usability, security and reliability. Design and implementation issues that could bring more flexibility to the framework are also treated in this step. Some pattern languages contain patterns in several abstraction levels like architectural and design patterns. Thus, it is possible that the non-functional aspects of the domain may have already been covered by the patterns. As our approach is preferably used with analysis pattern languages, this step was added to reinforce the need of analyzing these issues.





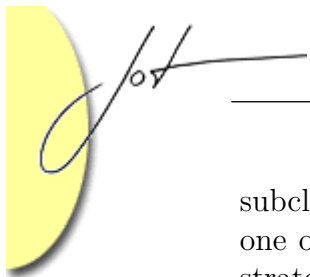
It is necessary to balance performance versus flexibility when considering non-functional requirements and design or implementation issues, because including several alternatives in the framework would make it more flexible but degrade system performance. For example, if we consider database portability, the choice of using a relational database or an object-oriented one may derive a hot spot to be set by the framework user. Another example is the graphical user interface, which could have two or more implementations (one for traditional applications, another for virtual applications, etc.) so that the framework user could choose one of them. Notice that this type of flexibility could be achieved by implementing several versions of the framework, which would cause less impact on system performance. An example of a design/implementation issue that could generate a hot spot is a web-based education framework, where the course selection mechanism can vary [Fontoura et al., 2001]. For example, the entire list of available courses or just the ones related to the student major could be shown.

Depending on the framework developer knowledge and on human decisions made at this point, other hot spots that are not explicit in the pattern language can be added. Knowledge about the domain is essential to succeed, but some guidelines can give the framework developer indications of other sources of hot spots. For example, looking at class attributes in the patterns of the pattern language, some questions should be answered, like: “is this a computed attribute?” If so, “is it possible to have several types of algorithms to compute it?” If the answer is affirmative, a new hot spot has been found. Looking at class relationships, another source of hot spots is to argue the cardinality of the relationships. If it is possible to find applications where the cardinality would be different from the cardinality proposed in the pattern, then a variant of the pattern exists and, consequently, a new hot spot. It is also desirable to look for similar hot spots in the table, because sometimes new hot spots can be derived by analogy.

Applying this process results in a list of the framework hot spots, each of which composed of an identifier code, a description, a type, the section(s) in the pattern where this hot spot was found and the pattern number (these two last items are optional). The hot spot type allows the framework developer to know what should be done to obtain an application from the framework. For example, if the type is `PARTIC_CHOICE` then there will be a choice among participants; if the type is `PATTERN_OPTION`, then the pattern is optional in particular applications; and if the type is `PARTIC_OPTION`, then a pattern participant can be omitted in particular instantiations.

## 5 DESIGN OF THE FRAMEWORK HOT SPOTS

According to the hot spot type (Table 1) it is possible to have some indications of how to implement it in the framework. For example, when a choice has to be made among participants (`PARTIC_CHOICE` type), the `STRATEGY` design pattern [Gamma et al., 1995] can be used. A Strategy class can be created with concrete



subclasses representing each possible choice. This strategy class is referenced by one of the pattern mandatory participants, whose operations are delegated to the strategy object.

Another example of how the hot spot type can influence its design is for optional patterns (PATTERN\_OPTION), in which the design has to allow the omission of its constituent classes during instantiation, when that pattern is not selected. It is preferable to design the classes using only inheritance, as the simple fact of inheriting by one of the subclasses is enough to know that the pattern was used. However, this is not always possible. For example, when the application of a pattern opens the possibility of using another dependent pattern, then it may be necessary for a class of the first pattern to know if the dependent pattern was also applied. A possible solution to solve this problem is to include a special class variable in the first pattern class that indicates if the dependent pattern was applied.

Similarly, for optional participants (PARTIC\_OPTION), there must be a way of hiding them during instantiation. This can also be done by including a special class variable in a mandatory class of the pattern to indicate if an optional participant was used in a specific instantiation.

The other three types of hot spots, namely BEHAVIOR, RELATIONSHIP, and PROPAGATION usually do not map easily to a common design, as they are more general. So, each specific case has to be treated separately.

## 6 CASE STUDY

A case study was conducted to evaluate the proposed process. The GRN pattern language [Braga et al., 1999] was used to develop a framework for the same domain, called GREN [Braga and Masiero, 2002b].

### Pattern Language features

The Pattern Language for Business Resource Management (*Gestao de Recursos de Negocios*, or GRN, in Portuguese) is composed of fifteen analysis patterns (see Figure 2), some of which are specific usages or extensions of more generic patterns proposed in the literature [Coad et al., 1997, Boyd, 1998, Johnson and Woolf, 1998, Fowler, 1997]. It was conceived to help software engineers in the development of applications concerned with business resource management. This includes applications where it is necessary to log transactions of business resource rental, trade or maintenance. By transaction we mean the same as Coad et al.: “a significant event to be remembered, i.e., an event that the system must remember through time” [Fowler, 1997].

Resource rental focuses primarily on the satisfaction of a certain temporary need of a product or service like a videotape or a physician time. Resource trade focuses

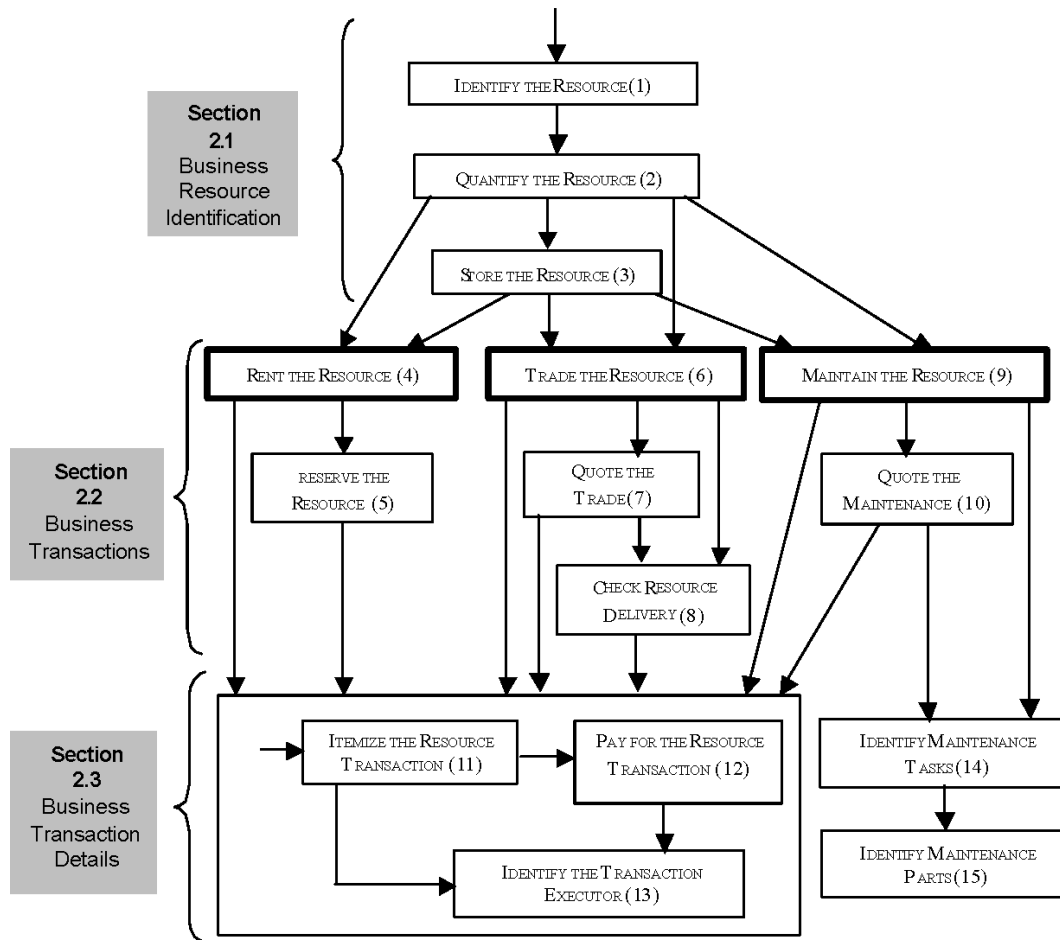


Figure 2: GRN: a Pattern Language for Business Resource Management

on the transference of property of a product, as for example a product sale or auction. Resource maintenance focuses on the maintenance of a certain product, using labor and parts to perform it, as in an electric appliance repair shop. Figure 2 shows the dependencies among the patterns and the order in which they are generally applied. These dependencies are also presented, and eventually complemented, inside each specific pattern. The main patterns in the language are **RENT THE RESOURCE**, **TRADE THE RESOURCE**, and **MAINTAIN THE RESOURCE**, indicated by a thicker line. Their use is not mutually exclusive and, in fact, there are applications in which they can fit together. **MAINTAIN THE RESOURCE** may use **RENT THE RESOURCE** and **TRADE THE RESOURCE**, as in a car repair shop system, in which parts are traded and labor is rented.

The patterns are grouped according to their purpose, as illustrated in Figure 2: group 1 patterns are basically concerned with the identification, quantification and storage of the business resources; group 2 patterns deal with the business transactions performed by the system; and group 3 patterns that take care of de-

tails associated to most business transactions. The GREN framework construction based on the GRN pattern language was done in two phases. In the first phase a white box version of the framework was built [Braga and Masiero, 2002b]. Its hot spots were identified in the GRN pattern language and were implemented using Smalltalk VisualWorks. This framework can be manually instantiated using a well defined process [Braga and Masiero, 2002c]. In the second phase, a wizard was built [Braga and Masiero, 2002a], also based on the GRN pattern language, to automatically instantiate the framework to specific applications.

## Application of the hot spots identification process

The process outlined in section 4 was applied to identify 36 hot spots of the GREN framework (some of them are listed in Table 2). Most of them (88,9%) were found based in the GRN pattern language (steps 1 and 2 of the process) and only 11,1% were identified by other ways (step 4 of the process, in this case).

Table 2: Partial List of Hot Spots for the GREN Framework

#	Name	Description	Type	Source in the pattern language	Pattern #
2	Resource Quantification	A resource can be unique, can have multiple instances, can be managed in quantities or in lots.	PARTIC_CHOICE	Participants, Structure, Variants (sub-patterns)	2
8	Instance number generation	The instance number of the resource to be rented can be supplied by the user or automatically generated by the system	BEHAVIOR	-	4
9	Resource reservation	The application may or may not need to deal with resource reservation	PATTERN_OPTION	Language Graph Context +	5
12	Reserved quantity entry	It is necessary to read the quantity of reserved resources when the reservation refers to a measurable resource (propagation of pattern 2 usage)	PROPAGATION	-	5
13	Resource trade	The application may or may not concern the trade of resources	PATTERN_OPTION	Language Graph Context +	6
14	Existence of Source-party in the trade	In sale systems, the organization may be small and not have branches or departments.	PARTIC_OPTION	Participants	6
15	Existence of Destination-party in the trade	In purchase systems, the organization may be small and not have branches or departments.	PARTIC_OPTION	Participants	6
16	Traded quantity entry	It is necessary to read the quantity of traded resources when the trade refers to a measurable resource (propagation of pattern 2 usage)	PROPAGATION	Participants	6
35	Charge of rentals	The application may have no charge for rentals, as in some libraries	BEHAVIOR	-	4
36	Fine computing	The computing of fines due to delayed payment may vary from application to application	BEHAVIOR	-	12

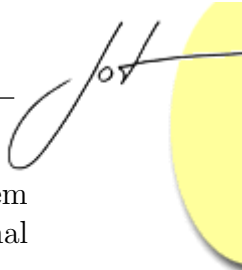


Table 3 summarizes the types of hot spots found. We observe that most of them are of type PATTERN\_OPTION, since the pattern language has many optional patterns and this results directly in hot spots, as explained in section 3.

Table 3: Summary of the types of hot spots identified

Type of hot spot	Quantity found					Percentage
	Step 1	Step 2	Step 3	Step 4	Step 5	
PATTERN_OPTION	14					38,9 %
PARTIC_OPTION		7				19,4 %
PARTIC_CHOICE		3				8,3 %
RELATIONSHIP		1				2,7 %
BEHAVIOUR		2		3		13,9 %
PROPAGATION		5		1		16,7 %
<b>Total</b>	<b>36</b>					

To illustrate the hot spots identification based on the pattern language, we will use one pattern of the GRN pattern language, shown in Figure 3. It refers to the TRADE THE RESOURCE pattern. The four hot spots identified from this pattern correspond to hot spots 13 to 16 of Table 2. Hot spot 13, called “Trade the Resource”, provides the flexibility to make optional the application of this pattern, because the pattern language can also be used for rental or maintenance applications, and so it may be desired not to apply this pattern. This is a PATTERN\_OPTION hot spot, as the whole pattern is optional. It was identified during step 1 of the process of Figure 1. Analyzing the pattern language graph (see Figure 2), we notice that there are paths that do not include the TRADE THE RESOURCE pattern, which implies that this pattern is optional. Also, observing the pattern context (see item 6.1 of Figure 3), we notice that applications that do not deal with resource trade do not fit in the proper context for the pattern usage and, thus, should not use it.

Hot spots 14 to 16 were found during step 2 of the process shown in section 4. Hot spot 14, called “Source-party existence”, gives small organizations the possibility of having a simpler system, in which branches or departments are not considered. This hot spot was identified observing the “participants” section of the pattern (see item 6.5 of Figure 3: notice that “Source-party” is an optional participant of this pattern). It is a PARTIC\_OPTION hot spot, as its purpose is to make the source-party participant optional. Hot spot 15 (“Destination-party existence”) is similar to number 14 and was identified in the same section.

Hot spot 16 (“Traded quantity entry”) concerns the inclusion of a new attribute in the trade due to the previous use of another pattern of the language and, so, is a PROPAGATION hot spot. The knowledge about the propagation caused by the application of certain patterns is embedded in the pattern language, as exemplified by the participant “Resource Trade” (see section 6.5 of Figure 3). The pattern language states that an attribute `quantity` is included in this class when the MEASURABLE RESOURCE sub-pattern has been applied earlier.

**Pattern 6: TRADE THE RESOURCE**

**6.1 Context**  
Your application deals with trade of resources, which may involve resources sold and/or purchased. You have already identified and Quantified these resources. Resource trading may be thought of as a resource property transference, in which a resource owned by one party becomes owned by another party. In a sale, if the resource is not available in stock, then the customer can fill in an order that will be granted when possible. In a purchase an order is made to the supplier who delivers the resource within a certain period.

**6.2 Problem**  
How do you manage the resource trades made by your application?

**6.3 Forces**

- It is essential to log trade information, because it can be used to generate important reports on resource demand and organization gains (most systems in this domain are concerned with profits).
- The additional storage space and processing time required to log trade information has to be balanced against possible gains in system functionality when evaluating costs versus benefits. For example, it may be enough to increase and decrease stock levels when resources are traded, without considering other trade details.

**6.4 Structure**  
Figure 17 shows the TRADE THE RESOURCE pattern.

**Figure 17: TRADE THE RESOURCE pattern**

**6.5 Participants**  
**Resource Trade:** represents all the details involved in trading the resource. The attribute `status` denotes the trade stage: pending, partially fulfilled, or fully fulfilled. When the MEASURABLE RESOURCE sub-pattern has been applied earlier, then an attribute `Quantity` is added to denote a non-unitary resource trade. **Resource/Resource Instance/Resource Lot:** the choice among Resource, Resource Instance or Resource Lot depends on the quantification sub-pattern used. **Source-Party:** represents the original resource owner, for example, in the case of a sale it is the organization department or branch that sells the resource, and in the case of a purchase it is the supplier organization. This class is optional for small sale systems where there are no departments or branches. **Destination-Party:** represents the final resource owner, for example, in the case of a sale it is the customer buying the resource, and in the case of a purchase it is the organization department or branch buying the resource. This class is optional for small purchase systems where there are no departments or branches and also in systems where the customer is not logged, as in supermarkets.

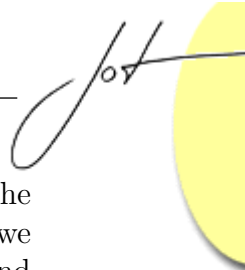
**6.6 Example**  
Figure 18 shows an instantiation of the TRADE THE RESOURCE pattern for an Inventory Control system.

**Figure 18: Instantiation of the TRADE THE RESOURCE pattern**

**6.7 Following patterns**  
Now, look at patterns in Section 2.3, which are useful for modeling other trade details. As a trade is followed by a delivery and can be preceded by a quotation, try to use the patterns QUOTE THE TRADE (7) and CHECK RESOURCE DELIVERY (8).

Figure 3: Example of a pattern of the GRN pattern language

Another interesting result of the case study was the identification of four new hot spots during step 4 of the process proposed in section 4. For example, hot spot 12 of Table 2 was not identified from the pattern language, but found through inspection of similar hot spots in the table. An analysis of the table was done to check the propagation effect of applying the QUANTIFY THE RESOURCE pattern. This pattern has four alternative solutions, presented as sub-patterns. One of them, the MEASURABLE RESOURCE sub-pattern, is used when the resource is dealt with in quantities, so the `quantity` attribute needs to be entered in all resource transactions. Thus, as we have seven possible transactions in the pattern language, we expected



to have seven hot spots concerning this feature. However, we found only four. The missing ones were for patterns 5, 9 and 10. Making a deeper domain analysis, we concluded that this feature is not desired for maintenance systems (patterns 9 and 10), because it is very rare to have resources to be maintained in quantities (they are usually unique). So only one new hot spot was added (number 12), as it makes sense to reserve more than one copy of a resource. It had been forgotten during the pattern language writing and, afterwards, the pattern language was fixed to include this requirement.

Hot spot 36 of Table 2 is another example of a hot spot found during step 4 by analyzing class attributes that need to be computed. There are several different ways of computing the fine that customers need to pay when a transaction is paid after its due date. Examples are fixed, daily, weekly or monthly fees, or a percentage of the total due value. Besides the fine, interests may optionally be charged. Hot spots 8 and 35 were also found during the fourth step of the process. It was not possible to identify hot spots during step 5 using the GRN pattern language because GRN refers to analysis patterns and is not concerned with design and implementation issues. We have decided not to include hot spots of this category in a first version of the framework, but will consider this possibility in its future versions.

## Design of the GREN hot spots

The GREN framework hot spot 2 – Resource Quantification (see Table 2) was implemented using the STRATEGY design pattern [Gamma et al., 1995]. Figure 4 shows part of the GREN class hierarchy, where it can be observed the use of a strategy object (*QuantificationStrategy* of Figure 4). This strategy object is responsible for the resource quantification issues, allowing the framework to implement the four different solutions required by the pattern. So, for example, when the method `isAvailable` is invoked by a *Resource* object (this method returns a boolean to denote if the resource is available for the transaction), it is delegated to the corresponding strategy object, which knows exactly how to determine the resource availability. For example, if the quantification strategy is an *InstantiableResource*, then it has to find at least one available instance, but if it is a *MeasurableResource*, then its quantity in stock needs to be checked.

Another example of a GREN hot spot design and implementation is hot spot 9 – Reserve the Resource, now illustrating the PATTERN\_OPTION type. It uses the class variable `hasReservation` in the *ResourceRental* class to indicate whether or not the RESERVE THE RESOURCE pattern was used. As can be seen in Figure 3, this pattern is optional, as there are arrows indicating that other patterns can be applied after pattern 4, so skipping it. This class variable is set to a boolean value during GREN instantiation for a specific application. If it is set to `true`, then pattern 5 was applied. As there is a relationship between a class of pattern 4 to a class of pattern 5, it is necessary to override some methods that deal with this relationship. Otherwise, pattern 5 was not applied, so nothing else has to be done.

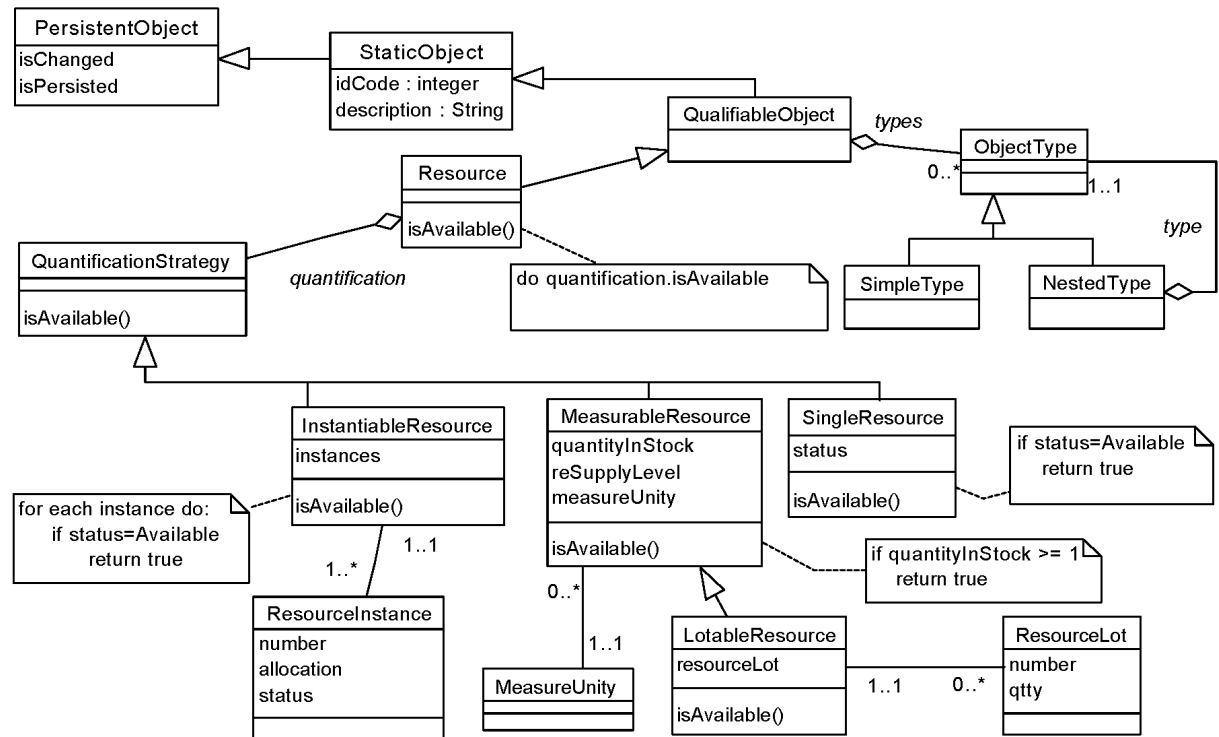


Figure 4: Example of some GREN classes

## 7 CONCLUDING REMARKS

It is rather intuitive that splitting the application domain in several patterns implies in the elicitation of many hot spots. This makes the system composed of smaller parts, which need to be joined to make up the specific application. This fact was confirmed during the case study, where we have observed many hot spots identified from the pattern language graph, which means that optional parts of the system are encapsulated in patterns.

The pattern language example used in this paper had only analysis patterns, so the hot spots found using it were basically concerned with the analysis phase of the software development. Pattern languages that comprise patterns in several abstraction levels, for example, architectural, analysis, and design patterns, can lead to the identification of other types of hot spots.

The pattern language construction involves an analysis of the application domain and having practical experience in the development of applications for this domain. In our case the first author had more than ten years of practice that allowed the development of the pattern language. During the framework usage in the development of specific applications and because of the application domain evolution, new hot spots may be necessary. In this case, the pattern language must also be updated, including new patterns or changing existing ones.





The strategy proposed in this paper offers a new alternative to find framework hot spots. The pattern language embodies the domain knowledge and lends itself to hot spots elicitation. As already mentioned in the introduction, our approach is directed to the ultimate goal of easing framework reuse and continues using the pattern language in the framework design, implementation and instantiation.

## REFERENCES

- [Aarsten et al., 2000] Aarsten, A., Brugali, D., and Menga, G. (2000). *A CIM Framework and Pattern Language*, pages 21–42. Domain-Specific Application Frameworks: Frameworks Experience by Industry, M. Fayad, R. Johnson, –John Willey and Sons.
- [Boyd, 1998] Boyd, L. (1998). *Business Patterns of Association Objects*, pages 395–408. Addison-Wesley.
- [Braga et al., 1999] Braga, R. T. V., Germano, F. S. R., and Masiero, P. C. (1999). A pattern language for business resource management. In *6th Pattern Languages of Programs Conference (PLoP'99)*, Monticello – IL, USA.
- [Braga and Masiero, 2002a] Braga, R. T. V. and Masiero, P. C. (2002a). GREN-Wizard: a tool to instantiate the GREN framework. In *Proceedings of the Tools Session of the 16th Simposio Brasileiro de Engenharia de Software (SBES 2002)*, pages 408–413, Gramado-RS.
- [Braga and Masiero, 2002b] Braga, R. T. V. and Masiero, P. C. (2002b). A process for framework construction based on a pattern language. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 615–620, IEEE Computer Society, Oxford-England.
- [Braga and Masiero, 2002c] Braga, R. T. V. and Masiero, P. C. (2002c). The role of pattern languages in the instantiation of object-oriented frameworks. *Lecture Notes on Computer Science*, 2426-Advances in Object-Oriented Information Systems:122–131.
- [Braga and Masiero, 2003] Braga, R. T. V. and Masiero, P. C. (2003). Building a wizard for framework instantiation based on a pattern language. *Lecture Notes on Computer Science*, 2817-Object-Oriented Information Systems, 2003: 95–106.
- [Brown and Whitenack, 1996] Brown, K. and Whitenack, B. G. (1996). *Crossing Chasms: A Pattern language for Object-RDBMS Integration, The Static Patterns*, pages 227–238. Addison-Wesley. in J. Vlissides and J. Coplien and N. Kerth (eds.)- *Pattern Languages of Program Design 2*.

- [Brugali and Menga, 1999] Brugali, D. and Menga, G. (1999). Frameworks and pattern languages: an intriguing relationship. *ACM Computing Surveys*, 32(1):2–7.
- [Brugali et al., 2000] Brugali, D., Menga, G., and Aarsten, A. (2000). *A Case Study for Flexible Manufacturing Systems*, pages 85–99. Domain-Specific Application Frameworks: Frameworks Experience by Industry, M. Fayad, R. Johnson, –John Willey and Sons.
- [Coad et al., 1997] Coad, P., North, D., and Mayfield, M. (1997). *Object Models: Strategies, Patterns and Applications*. Yourdon Press, 2 edition.
- [Coplien, 1998] Coplien, J. O. (1998). *Software Design Patterns: Common Questions and Answers*, pages 311–320. Cambridge University Press. in L. Rising - The Patterns Handbook: Techniques, Strategies, and Applications.
- [Fayad and Schmidt, 1997] Fayad, M. and Schmidt, D. C. (1997). Object-oriented application frameworks. *Communications of the ACM*, 40(10).
- [Fontoura et al., 2001] Fontoura, M., Pree, W., and Rumpe, B. (2001). *The UML Profile for Framework Architectures*. Addison-Wesley.
- [Fowler, 1997] Fowler, M. (1997). *Analysis Patterns*. Addison-Wesley.
- [Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.
- [Griss et al., 1998] Griss, M. L., Favaro, J., and d’Alessandro, M. (1998). Integrating feature modeling with the RSEB. In *Fifth International Conference on Software Reuse*, pages 78–85, IEEE Computer Society, Los Alamitos, CA, USA.
- [J. Bosch and Fayad, 1999] J. Bosch, P. Molin, M. Mattsson. P. Bengtsson. and M. Fayad (1999). *Framework problem and experiences*, pages 55–82. Building Application Frameworks: Object-Oriented Foundations of Framework Design, M. Fayad, R. Johnson, D. Schmidt, – John Willey and Sons.
- [Johnson and Foote, 1988] Johnson, R. and Foote, B. (1988). Designing reusable classes. *Journal of Object Oriented Programming*, 1(2):22–35.
- [Johnson, 1992] Johnson, R. E. (1992). Documenting frameworks using patterns. In *OOPSLA ’92*, pages 63–76.
- [Johnson, 1996] Johnson, R. E. (1996). *Transactions and Accounts*, pages 239–249. Addison-Wesley. in VLISSIDES, J.; COPLIEN, J.; KERTH, N. (eds.) Pattern Languages of Program Design 2.



- [Johnson and Woolf, 1998] Johnson, R. E. and Woolf, B. (1998). *Type Object*, pages 47–65. Addison-Wesley. in Martin, R.C.; Riehle, D.; Buschmann, F. Pattern Languages of Program Design 3.
- [Meszaros, 1995] Meszaros, G. (1995). *A Pattern Language for Improving the Capacity of Reactive Systems*, pages 575–591. Addison-Wesley. in J. Coplien and D. Schmidt (eds.) - Pattern Languages of Program Design.
- [Pree, 1995] Pree, W. (1995). *Design Patterns for Object-Oriented Software Development*. Addison-Wesley.
- [Pree, 1999] Pree, W. (1999). *Hot-spot-driven Development*, pages 379–393. Building Application Frameworks: Object-Oriented Foundations of Framework Design, M. Fayad, R. Johnson, D. Schmidt, –John Willey and Sons.
- [Roberts and Johnson, 1998] Roberts, D. and Johnson, R. (1998). *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks*, pages 471–486. Pattern Languages of Program Design 3, Martin, R.C., Riehle, D. , Buschmann, F. – Addison-Wesley.
- [Schmid, 1997] Schmid, H. A. (1997). Systematic framework design by generalization. *Communications of the ACM*, 40(10):48–51.
- [Schmid, 1999] Schmid, H. A. (1999). *Framework Design by Systematic Generalization*, pages 353–378. Building Application Frameworks: Object-Oriented Foundations of Framework Design, M. Fayad, R. Johnson, D. Schmidt, –John Willey and Sons.
- [Svahnberg et al., 2001] Svahnberg, M., Gorp, J. V., and Bosch, J. (2001). On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, pages 45–54, Amsterdam, The Netherlands.

## ABOUT THE AUTHORS



**Rosana T. Vaccare Braga** is Professor at ICMC – University of São Paulo – Brazil, where she had also had her PhD in 2003 and her Master’s degree in 1998. She has been working with business systems development for the last fifteen years. Her research activity is in Software Engineering (Frameworks, Pattern Languages, Reengineering, and Reverse Engineering) and Information Systems. Her financial support for this paper is from FAPESP Process n. 98/13588-4. She can be reached at [rtvb@icmc.usp.br](mailto:rtvb@icmc.usp.br).



**Paulo C. Masiero** received the D.Sc. degree from the University of São Paulo, São Paulo. He is currently a Full-Professor in the Computer Science Department of the Institute of Mathematics and Computing Sciences/USP, in São Carlos, Brazil, where he has been the Dean. His research activities are in Software Engineering (software reuse, object oriented design, frameworks, and reverser engineering) and applications of Information Technology. He can be reached at [masiero@icmc.usp.br](mailto:masiero@icmc.usp.br).