The Role of Pattern Languages in the Instantiation of White-Box Object-oriented Frameworks

Rosana T. V. Braga[*] and Paulo Cesar Masiero[**]

Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - Brazil
{rtvb,masiero}@icmc.sc.usp.br

Abstract In this paper we propose the use of pattern languages to guide a framework instantiation. Both the framework and the pattern language refer to the same domain, and the framework must have been constructed based on the pattern language. The framework instantiation here proposed is done in several steps: analysis of the specific application, mapping between the application and the framework, implementation of the application classes, and test of the final application. All these activities are supported 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. The proposed approach is illustrated with the example of a framework we have built based on a pattern language.
Keywords: Software reuse, frameworks, pattern languages, framework instantiation.

1 Introduction

Software patterns and pattern languages aim at reuse in high abstraction levels. Software patterns try to capture the experience acquired during software development and synthesize it in a problem/solution form [1]. A pattern language is a structured collection of patterns that build on each other to transform needs and constraints into an architecture [2]. A pattern language organizes the knowledge about a specific domain into specific patterns, that can be systematically applied in the development of systems for the same domain. It represents the temporal sequence of decisions that led to the complete design of an application, so it becomes a method to guide the development process [3].

Software reuse can be achieved by several means, among which are class libraries, software patterns and object-oriented software frameworks (from now on called simply frameworks). 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

[*] Financial support from FAPESP Process n. 98/13588-4.
[**] Financial support from FAPESP/CNPq.

from a single framework. However, frameworks are often very complex to build, understand, and use. Framework instantiation, which consists of adapting the framework to the specific application requirements, is complex and, most times, requires a complete understanding of the framework design and implementation details. According to Fayad and Johnson [4], the time to learn a framework in order to begin to use it can vary 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.

Pattern languages and frameworks can be associated to enhance software reuse. Pattern languages reflect experience in specific domains, covering all their main aspects. Consequently, they have built-in information about the points that differ from one application to another, acting as an excellent source for the identification of the framework hot spots [5]. Moreover, a pattern language can also be used for documenting the framework, as already shown in several works [6,7]; for supporting the framework design and implementation [8,3]; and as a method to guide the transformation of the framework in a concrete application [8]. Thus, the availability of a pattern language for a specific domain and its corresponding framework imply that new applications do not need to be built from scratch, because the framework offers the reusable implementations of each pattern of the pattern language. Therefore, the application development process may follow the language graph, from root to leaves, deciding on the use of each specific pattern and reusing its implementation offered by the framework. Although several researchers have noticed the relationship between pattern languages and frameworks, no work exists, to the author's knowledge, to show how to take advantage of this fact.

In this work we show how domain specific pattern languages can be used in the instantiation of a framework for the same domain, making the instantiation process easier and more systematic. The paper is organized as follows. Section 2 gives an overview of our approach and presents a running example. Section 3 presents the first step of the instantiation, which consists of the system analysis guided by the pattern language. Section 4 shows how to map the resulting analysis model to the framework. Section 5 gives some advices about the implementation of the specialized classes, as our proposal is aimed at white-box framework instantiation. Section 6 provides information to properly test the resulting application. Section 7 presents the concluding remarks.

2 Framework instantiation

Several approaches have been proposed to help in framework instantiation. They rely on the framework documentation to obtain the necessary information to instantiate an application, which basically consists of the framework class hierarchy, the abstract classes that need to be subclassed in the new application, the methods to be overridden in these classes, and examples of applications derived from the framework. There are at least four types of approaches for framework instantiation: framework documentation, exploration of exemplars, patterns, and cookbooks. The first consists of studying the framework documentation, i.e., its class hierarchy, source code, and other documents. Conventional training or special tutorials are ways of achieving this. The drawbacks of this approach are the time required to learn the framework and the difficulty to determine if the level of comprehension is enough to begin to use the framework.

The exploration of exemplars consists of examining existing applications built with the framework to identify what needs to be adapted to obtain the final system. The drawbacks of this approach are: the difficulty to find a similar application among the examples; what to do when a particular functionality is not found; what to do when a particular functionality is present in an example, but with additional features that are not needed; and the difficulty to know when to consider the instantiation done. An example of this approach is given by Gangopadhyay and Mitra [9].

Patterns can be used to document the framework and to show how to use it. Three related works have been done in this line [10,8,7], but they did not proceed towards the definition of an approach to be used by other framework developers. This paper explores this idea and tries to support framework development and instantiation based on pattern languages.

Cookbooks can be elaborated as a set of tasks needed to adapt a framework to a specific application, like in a recipe. Several works have been done in this line [11,12], but there are limitations like little flexibility, difficulty in finding the correct recipe, and because it is unlikely that these tasks can be done step by step, i.e., a choice made during the specialization process may change the remaining "recipe". Our work also uses a well defined process to guide the instantiation.

2.1 Framework instantiation based on a Pattern Language: an overview

Our approach uses a pattern language to support framework instantiation, as illustrated in Figure 1. It consists of four steps: system analysis, mapping between analysis model and framework, implementation of specific classes, and test of the resulting system. Each step is explained in detail in sections 3 to 6. The framework must have been built based on the same pattern language using our general process [13,14], summarized in section 2.2.

2.2 Framework Construction based on a Pattern Language

To construct a framework from a pattern language we first need to develop a pattern language for a specific domain. The experience acquired during software development in a domain or reverse engineering of existing systems can be used to accomplish this. Details of how to construct the pattern language are out of the scope of this paper, but basically it involves three steps: domain analysis, splitting the problems found in the domain into smaller problems, and creating a pattern to solve each of these problems.

The pattern language is then used to develop a white-box framework for that domain. The main goal of the framework is to allow code reuse for all classes

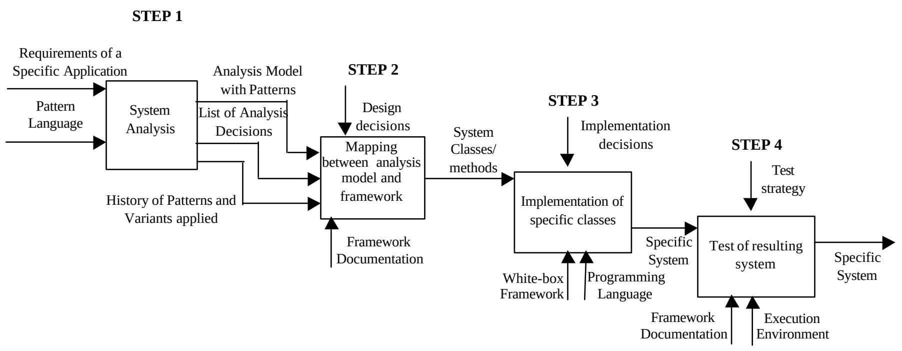


Figure 1. Framework Instantiation Process

belonging to the pattern language. We begin by identifying the framework hot spots, using information present in the pattern language [5]. Then, based on the resulting list of framework hot spots and on the pattern language, the framework is designed. 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 that can be used to design the framework part that takes care of the problem solved by the pattern. Finally, the framework classes defined in the framework design model are implemented, using a particular programming language.

A special documentation of the framework is done to ease its future 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. The idea is that 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.

Besides knowing which classes need to be subclassed, it is necessary to know which methods need to be overridden in the new classes. This is necessary because a white-box framework contains abstract classes whose behavior needs to be defined by the specialized classes, through methods called “hook methods” by Pree [15].

A wizard to help instantiating the framework may also be 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. A more detailed description of the approach for constructing a framework based on a pattern language can be found elsewhere [13].

2.3 Example of a pattern language and its associated framework

To illustrate our approach, we use the business resource management domain, for which we have built a pattern language, denominated GRN[16], and a white-box framework, denominated GREN[17]. At present we are building a tool to partially automatize the instantiation process. In this paper we only use the GREN white-box version.

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 video-tapes, 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. Figure 2 shows the relationships among the patterns. 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 3 shows the **Maintain the Resource** pattern, extracted from the GRN pattern language [16]. Observe that the pattern structure diagram uses the UML notation with some modifications. We have included special markers before input and output system operations, which are more than methods, as they are executed in response to system events that occur in the real world. We use "?" for input operations that change the internal state of the system, and "!" for output operations, which generate system outputs without changing the system's state. Furthermore, a "#" before a method name means that its call message is sent to a collection of objects, instead of to a single instance, i.e., it will probably be implemented as a class method. It can be observed in the "Participants" section that this pattern has an optional participant, Source Party, which indicates a framework hot spot. Observe also the "Following Patterns" section, which guides the user to the next patterns to be used.

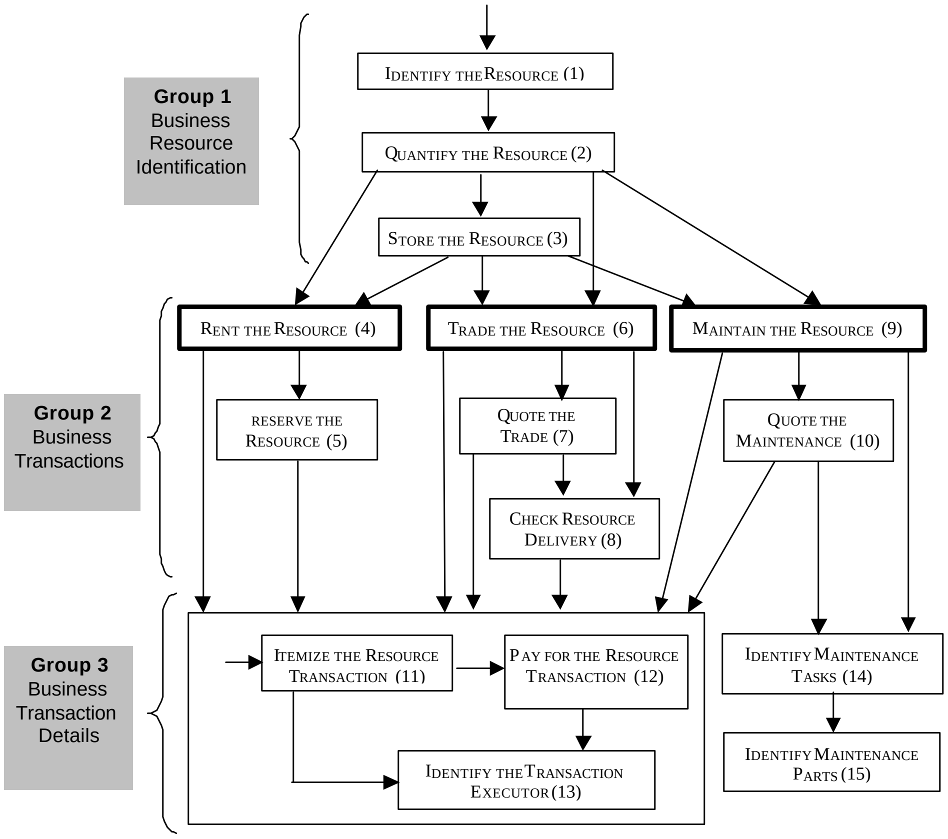
The GREN framework was developed to support the implementation of applications modeled using the GRN pattern language. All the behavior provided by classes, relationships, attributes, methods, and operations of GRN patterns is available on GREN. Its implementation was done using the VisualWorks Smalltalk [18] and the MySQL database [19] for object persistence. The first version of the GREN framework contains about 150 classes and 30k lines of code in Smalltalk. GREN was implemented using three layers: persistence, application, and GUI.

Tables 1 and 2 exemplify GREN documentation, specially developed to guide its instantiation for specific applications based on the usage of the GRN pattern language. Table 1 is used in the creation of the concrete classes of the application layer and Table 2 in the creation of the graphical user interface (GUI) classes. The application classes are concerned only with the system functionality. Each application class may have one or more corresponding GUI classes, concerned only with input/output data aspects. Notice that these tables are built based on the pattern language.

Table 1. Example of the GREN documentation - Table used to identify new application classes

Pattern	**Variant**	**Pattern class**	**GREN class**	**Ref code**
1-Identify the Resource	All	Resource	Resource	N1
	Default, Multiple types	Resource Type	SimpleType	N2
	Nested types	Resource Type	NestedType	N2
9 - Maintain the Resource	All	Resource Maintenance	ResourceMaintenance	N21
		Source Party	SourceParty	N6
		Destination-Party	DestinationParty	N14

GREN has special tables listing the methods to be overridden in the newly created classes, as exemplified in Table 3. These tables need to be followed sequentially, using information about the class hierarchy of the new application



Group 1 Business Resource Identification
Identify the Resource (1)
Quantify the Resource (2)
Store the Resource (3)
Group 2 Business Transactions
Rent the Resource (4)
Trade the Resource (6)
Maintain the Resource (9)
reserve the Resource (5)
Quote the Trade (7)
Quote the Maintenance (10)
Check Resource Delivery (8)
Group 3 Business Transaction Details
Itemize the Resource Transaction (11)
Pay for the Resource Transaction (12)
Identify the Transaction Executor (13)
Identify Maintenance Tasks (14)
Identify Maintenance Parts (15)

Figure 2. GRN Pattern Language: relationship among patterns

Pattern 9: MAINTAIN THE RESOURCE

Context

Your application deals with resource maintenance or repair. You have already identified and quantified these resources, which are basically customer assets that present faults or need periodic maintenance. They must be fixed to be used again or to prevent them from failing within a time interval. For example, cars, television sets, electric appliances, and computers are resources that often have problems during their life cycle.

Problem

How do you manage resource maintenance performed by your application?

Forces

- Keeping maintenance records is important both to customers and to organizations that do maintenance. Customers have the right to complain if the maintenance is not satisfactory. Parties usually need this information for financial purposes. A simple alternative when this information does not need to be kept is to have a resource attribute containing the last maintenance date.
- Extra space is needed to store maintenance information, and having several maintenance records related to each resource implies more processing time to retrieve the last maintenance.

Structure

Figure 18 shows the MAINTAIN THE RESOURCE pattern.

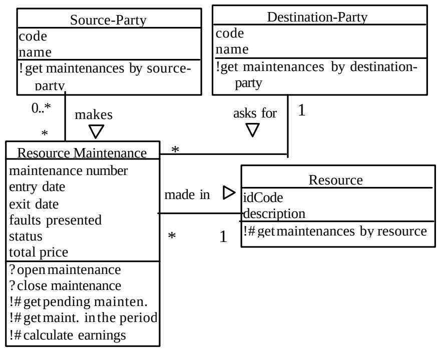


Figure 18: **MAINTAIN THE RESOURCE pattern**

Participants

Resource Maintenance: represents all the details involved in maintaining a resource. The attribute `faults presented` describes what is wrong with the resource. The method `open maintenance` is used to register the resource to be repaired; `close maintenance` is executed when the maintenance finishes, and `get pending maintenances` retrieves maintenances that are not finished. The attribute `status` denotes the maintenance stage: pending, partially fulfilled, or fully fulfilled.

Resource: as described in previous patterns.

Source-Party: represents the department or branch of the organization that is responsible for doing the maintenance. This class is optional in this pattern because in small organizations, with no branches or departments, it is not worth creating it.

Destination-Party: represents the owner of the resource being maintained as, for example, the customer.

Example

Figure 19 shows an instantiation of the MAINTAIN THE RESOURCE pattern for a Car Repair Shop system.

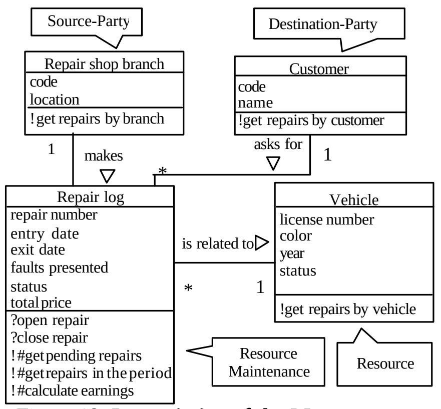


Figure 19: **Instantiation of the MAINTAIN THE RESOURCE pattern**

Following patterns

Check now the other patterns in Section 2.3, which deal with other maintenance details After, check the convenience of using the QUOTE THE MAINTENANCE (10) and the IDENTIFY MAINTENANCE TASKS (14) patterns.

Figure 3. Maintain the Resource pattern

Table 2. Example of the GREN documentation - Table used to identify new GUI classes

Pattern	**Variant**	**Ref code**	**GREN class**
1-Identify the Resource	Single Resource	N1	SingleResourceForm
	Measurable Resource	N1	MeasurableResourceForm
	Instantiable Resource	N1	InstantiableResourceForm
	Default, Multiple Types	N2	StaticObjectForm
	Nested type	N2	QualifiableObjectForm
9 - Maintain the Resource	With application of patterns 14 and 15	N21	OneResourceMaintWPWTForm
	Without application of patterns 14 and 15	N21	OneResourceMaintNPNTForm
	All	N6	SourcePartyForm
	All	N14	DestinationPartyForm

classes to be instantiated. When a method is found that belongs to a GREN abstract class specialized during instantiation, that method is implemented in the specialized class. We show how to use these tables on Section 4.2.

Table 3. Examples of methods to be overridden

Super-class	**Method**	**Instance/Class**	**Comment**
QualifiableObject	typeClasses	C	returns a List with the classes that represent the resource type (zero or more).
ResourceMaintenance	resourceClass	C	returns the class that represents the resource being maintained.
	hasSourceParty	C	returns true if the participant "SourceParty" has been used during the instantiation and false otherwise, as this participant is optional in the pattern.
	sourcePartyClass	C	returns the class name for the concrete class playing the role of SourceParty in the pattern. This method is only overridden if this participant was used during the GRN usage
	destinationPartyClass	C	returns the class name for the concrete class playing the role of DestinationParty in the pattern.

3 System Analysis

In this section we show how to use the pattern language to help in the analysis of a system in the same domain (step 1 of Figure 1). The input for this analysis is the requirements document for a specific system and the pattern language for the same domain. The result of this step is the analysis model of the system, the history of patterns and variants applied, and the list of analysis decisions made when the pattern language did not match the system requirements.

The example used in this paper to illustrate our approach is of a Pothole Tracking Repair System (PHTRS), whose requirements were established by

Pressman [20] and are reproduced here. "Citizens can log onto a Web site and report the location and severity of potholes. As potholes are reported they are logged withing a "public works department repair system" and are assigned an identifying number, stored by street address, size (on a scale of 1 to 10), location (middle, curb, etc.), district (determined from street address), and repair priority (determined from the size of the pothole), Work order data are associated with each porthole and includes pothole location and size, repair crew identifying number, number of people on crew, equipment assigned, hours applied to repair, hole status (work in progress, repaired, not repaired), amount of filler material used and cost of repair (computed from hours applied, number of people, material and equipment used). Finally, a damage file is created to hold information about reported damage due to the pothole and includes citizens name, address, phone number, type of damage, dollar amount of damage. PHTRS is an on-line system. All queries are to be made interactively."

3.1 General guidelines

Based on the specific system requirements, the pattern language is used to model the system. The pattern language should have been studied in advance by the developer, so that he or she knows if it can be applied in the analysis of this specific system. We assume that the pattern language contains analysis patterns, each of which containing at least the usual elements of a pattern, like problem, context, forces, and solution. In particular, the solution must contain a class diagram illustrating the pattern participants. It is also desirable to have a diagram showing the interaction among the patterns of the pattern language.

A pattern language usually has all the elements necessary for its usage. So, even an inexperienced developer needs only to read it to obtain all the guidance he or she needs to use it. However, we provide the following generic algorithm to help you using a pattern language.

1. Analyze each pattern and decide whether or not to use it. This analysis begin in the context section, where you find the elements that should be part of your application context in order to use the pattern. If this is the case, then analyze problem to check if it matches the problem if you have to solve in your system. Also look at the forces to make sure they reflect the restrictions, benefits and drawbacks you confront in your system.
2. If you decide to use the pattern, analyze the solution and possible variants or sub-patterns. This helps you to find the best solution to your specific system.
3. Sketch the class diagram referring to the portion of the system modeled by the pattern. Use a different color or symbol to highlight possible new attributes, methods or operations added to the pattern. This class diagram will grow in a gradual way, as new patterns are applied.
4. For each applied pattern, make a special mark in the requirements document to indicate which requirements were fulfilled.

5. For each applied pattern, use tags to indicate the roles played by each class in the pattern.
6. Make a summary table to inform, for each applied pattern, the pattern name, variant or sub-pattern used, and roles played by each class. This "history of patterns and variants applied" will be useful for the future framework instantiation, when you will need to figure out which framework super-classes to specialize. The wizard we are developing supports this activity and produces this table automatically.
7. After having applied the pattern language, check the requirements document to find non-attended or partially attended requirements. Complement the class diagram to fulfill them, by adding new attributes, classes, relationships, methods and operations. Remember to highlight these complements to distinguish them from the rest of the diagram. Take notes of the analysis decisions you make here, because they are essential in the future maintenance of the system. Annotate the requirements document to identify that these requirements are not covered by the pattern language.
   The result of this step is the system analysis model, the list of decisions made during analysis, and the history of patterns applied.

3.2 Example

The PHTRS requirements were analyzed using the GRN pattern language, producing its analysis model, shown in Figure 4. Tags were included to show the patterns used to model the system. A tag shows the role played by the class it points to. Its format is "P#n: role", where "n" is the pattern number and "role" is the role played by the class in that pattern. Notice that we do not represent canonic methods and operations.

During the analysis, a match has to be done between the pattern attributes, methods and operations versus the concrete class attributes, methods and operations, as recommended in the third step of Section 3.1 algorithm. For the PHTRS example, some additional attributes were added, as for example the attribute `numberOfPeopleOnCrew` (*WorkOrder* class).

Table 4 was built to show the seven patterns used during the instantiation, together with the roles played by each application class. During the application of pattern 1 - Identify the Resource, it was obvious that Pothole was the resource managed by the application. We considered Pothole as a multiple-typed resource, as we may need to list potholes by district, by size, by location or by citizen. When applying pattern 2 - Quantify the Resource, we decided that a pothole is unique, so we have chosen the Single Resource sub-pattern (other options, like resources that are measured, instantiated or dealt with by lots were not adequate in our case). After having skipped patterns 3 to 8, because our application was not concerned with resource storage, rental, reservation, trade, quotation or delivery, we have applied pattern 9 - Maintain the Resource - which allows resources to be maintained. We decided to assign the role of "Destination Party" to the "Public Works Department" and not to use the aspasSource Party participant, because the "Public Works Department" plays both roles in this case: it asks for a pothole repair and does the repair itself. Then, we have skipped patterns: 10, because the system does not concern maintenance quotation; 11, because each work order refers to only one pothole; and 12, because the system is not worried about payments associated to the work order. We have applied pattern 13 - Identify the Transaction Executor, as the system refers to the repair crew responsible for the work order. After that, we have applied pattern 14 - Identify the Maintenance Tasks, to deal with the several tasks associated with the work order, followed by pattern 15 - Identify the Maintenance Parts, which takes care of specifying the several materials used in the repair.

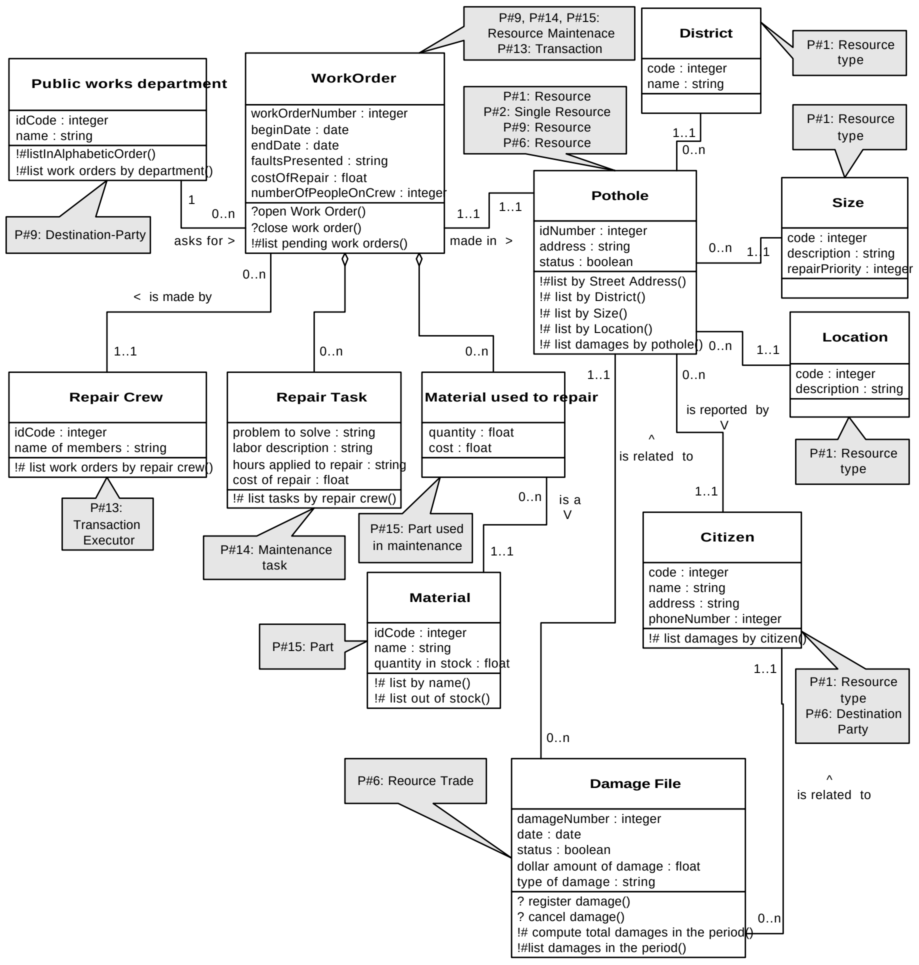


Figure 4. PHTRS Analysis Model with Patterns

Table 4. History of patterns used in the instantiation (TH)

Pattern	**Variant**	**Participant**	**Application Class**
1-Identify the Resource	Multiple types	Resource	Pothole
		Resource Type	District
		Resource Type	Size
		Resource Type	Location
		Resource Type	Citizen
2 - Quantify the Resource	Single Resource	Resource	Pothole
9 - Maintain the Resource	No source party	Resource	Pothole
		Resource Maintenance	Work Order
		Destination-Party	Public works department
13 - Identify the Transaction Executor	No commissions	Transaction Executor	Repair Crew
		Transaction	Word Order
14 - Identify Maintenance Tasks	Transaction executor instead of task executor	Resource Maintenance	Work Order
		Maintenance Task	Repair Task
15 - Identify Maintenance Parts	Default	Resource Maintenance	Work Order
		Part used in Maintenance	Material used to repair
		Part	Material
6 - Trade/Use the Resource	No source party	Resource	Pothole
		Resource Trade	Damage File
		Destination-Party	Citizen

At this point, the only requirement not attended by the framework was the damage file creation. If we analyze the semantic of the GRN patterns, we do not find a pattern to address this requirement, as we want to log the damages caused to citizens due to the pothole and this is not a rental, trade or maintenance transaction. However, if we analyze the patterns syntax, we see that the *Resource Trade* class has similar attributes to those we are looking for to our damage file. Like in a trade (sale or purchase), we need a date (the damage date), a status (was the dollar amount paid or not?), a value (the dollar amount), and some observation (the type of damage). Also, like in a trade, we need to link the damage file to a destination party (the citizen) and to a resource (the pothole). So, we decided to use pattern 6 - Trade the Resource, to model the damage file. This enabled us to use the GREN framework in a way not envisioned by its designers, which is called "flexing" or "domain abstraction" by Butler et al [21]. The pre-condition for flexing is that the pattern has the required computational structure, although it does not carry the domain semantics anticipated by the designer. We call this "semantic/syntatic replacement process", and note it in Table 4 by the general Use the Resource pattern.

The result of this step is the PHTRS analysis model (Figure 4) and the history of patterns used in the instantiation (Table 4). We do not have a list of analysis decisions made, because GREN supported all the functionality we needed for PHTRS. If we had included additional classes or relationships, not covered by GREN, then we would need to highlight this fact to ease future implementation and maintenance of the resulting system.

4 Mapping between the analysis model and the framework

In this section we show how to map the resulting analysis model to the framework (step 2 of Figure 1). The input for this mapping is the analysis model of the system, the history of patterns and variants applied, the list of analysis decisions made, and the framework documentation. We consider that the framework was documented according to the guidelines we suggest when building a framework based on a pattern language, i.e., the relation between the patterns of the pattern language and the framework classes is properly documented. The output of this step is a list of framework classes and corresponding methods to be implemented.

4.1 General guidelines

The procedure to be followed when mapping the analysis model to the framework is specific of each pattern language/framework pair. The following algorithm tries to abstract the most important activities to perform this mapping.

1. Consider the "history of patterns and variants applied" created during system analysis (item 6 of section 3.1) that contains, for each applied pattern, the variant or sub-pattern applied and the roles played by each class. Let us call this table TH. For each row of TH, let P be the applied pattern, V be the variant or sub-pattern applied, A be the application class, and R be the pattern class, so that A is playing the role R in pattern P/variant V.
2. Create a table that will contain the classes to be created in the new system. Let us call this table TC. This table has four columns: application class, class to create, framework super-class, and new attributes.
3. For each row of TH, check the framework documentation to find which classes are required in the new system for A, given the keys P, V and R. The result is the superclass names, of which A will inherit from. Create a row in TC for each new required class. There may be cases in which several classes are created corresponding to A (see example in Table 5). The new attributes' column is filled in with the attributes that were highlighted in the analysis model.
4. Consider now the resulting TC. For each row of TC, examine the framework documentation to identify which methods need to be overridden and define the contents of such methods. Also define the methods necessary to implement new attributes/operations, defining their functionality.

The result of this step is a list of new classes and methods to be implemented in the next step.

4.2 Example

The mapping between PHTRS and the GREN framework was done using the special documentation provided by the GREN framework. As mentioned in Section 2, this documentation consists of several tables showing, for each pattern of the GRN pattern language, the corresponding GREN classes to inherit from and the methods to be overridden during instantiation.

Based on Table 4 (TH), Table 1 of GREN documentation was used to identify the new application classes to be created, and Table 2 to identify the correct GUI form to be specialized. The result is partially shown in Table 5 (TC). Table 1 and Table 2 are used in the following way: Match the first three columns of TH with the first three columns of Table 1. The result is a GREN class to be specialized, so add a new row to TC, in which the application class is the fourth column of TH, the class to be created has the same name, and the GREN superclass is the fourth column of Table 1. Use the reference code (fifth column of Table 1) to search Table 2, using TH to match the appropriate GREN class, according to the pattern and variant applied. If a match is possible, then a new row is added to TC, in which the application class is the fourth column of TH, the class to be created has the same name with a special suffix to distinguish it from the application class (we have used the "Form" suffix in PHTRS), and the GREN superclass is the fourth column of Table 2. For example, the *Pothole* application class (ref code=N1) inherits from *Resource* and has a corresponding *PotholeForm* GUI class, inheriting from *SingleResourceForm*, because *Pothole* is a *SingleResource*. The *WorkOrder* application class (ref code=N21) inherits from *ResourceMaintenance* and has a corresponding *WorkOrderForm* GUI class, inheriting from *OneResourceMaintWPWTForm*, because patterns 14 and 15 were applied during instantiation.

Table 5. Classes to be created in the PHTRS system (TC)

Application Class	Class to create	GREN super-class	Additional Attributes
Pothole	Pothole (N1)	Resource	
	PotholeForm	SingleResourceForm	
Size	Size (N2)	SimpleType	repairPriority
	SizeForm	StaticObjectForm	
Work Order	WorkOrder (N21)	ResourceMaintenace	numberOfPeopleOnCrew
	WorkOrderForm	OneResourceMaintWPWTForm	
Citizen	Citizen (N14)	DestinationParty	address, phoneNumber
	CitizenForm	DestinationPartyForm	

The next phase is to examine the hook methods to be overridden in the newly created classes. For example, Table 3 shows a sample of a GREN table with the information about hook methods. In order to better understand the concept of hook methods, consider Figure 5, which shows part of the GREN class hierarchy. All classes that contain methods in *italics* are abstract classes, and these methods need to be overridden in specialized classes, so they appear in Table 3. In fact, some of them are optionally overridden, depending on the framework usage. For example, the method `hasSourceParty` of the abstract class *Business-ResourceTransaction* needs to be overridden, while the method `sourcePartyClass` is optional, depending on whether the "Source Party" participant of the pattern Maintain the Resource (Figure 3) was used. This information is not represented in the diagram, so we have decided to use a table to show the mapping and to give a better explanation of the method usage.

5 Implementation of the specific classes

In this section we give some advices about the implementation of the specialized classes (step 3 of Figure 1). The input for the implementation is the list of classes and methods obtained in step 2 and the programming language (the same used for the framework implementation). The output is the new application code.

5.1 General guidelines

The implementation of the new classes and corresponding methods require knowledge about the programming language in which the framework was developed. Some guidelines:

1. Begin by creating all the classes identified during the mapping, and their corresponding methods.
2. Create additional attributes and the necessary methods.
3. Create the GUI for the main window, together with a menu to be executed by the end user. This menu should contain shortcuts to all system operations.
4. Finally, compile the new application.

5.2 Example

The GREN documentation offers several algorithms used in the new classes creation, adaptation of the GUI to include new attributes, and production of the new system menus. Smalltalk VisualWorks must be used, which is the language in which GREN was implemented. The result is the specific application code, illustrated in Figure 5. The resulting number of lines of code for the PHTRS example was 1600 (1,6K LOC). It is important to observe that this new code refers only to the class declarations, overridden methods, and definition of new GUI forms (visually programmed). The GREN framework code, which is about 30K LOC, needs to be integrated to the final application, because it contains the basic superclasses that allow the system working.

Figure 5 illustrates some hook methods that were overridden in the PHTRS example. For example, the *Pothole* class inherits from *Resource*, which in turn inherits from *QualifiableObject*. So, according to Table 3, the `typeClasses` method needs to be overridden. See the comments column on Table 3 to know how these methods have been coded. It is also necessary to add methods to deal with new attributes added to the classes, as for example `repairPriority` in class

Size. In GREN, the cookbook defines which methods need to be overridden in this case: accessing methods (set and get), initialization methods, and placement of new widgets in the GUI form.

Figure 6 shows the main menus for PHTRS, where you can observe part of the functionality stated in the requirements (see section 2.2). All reports that

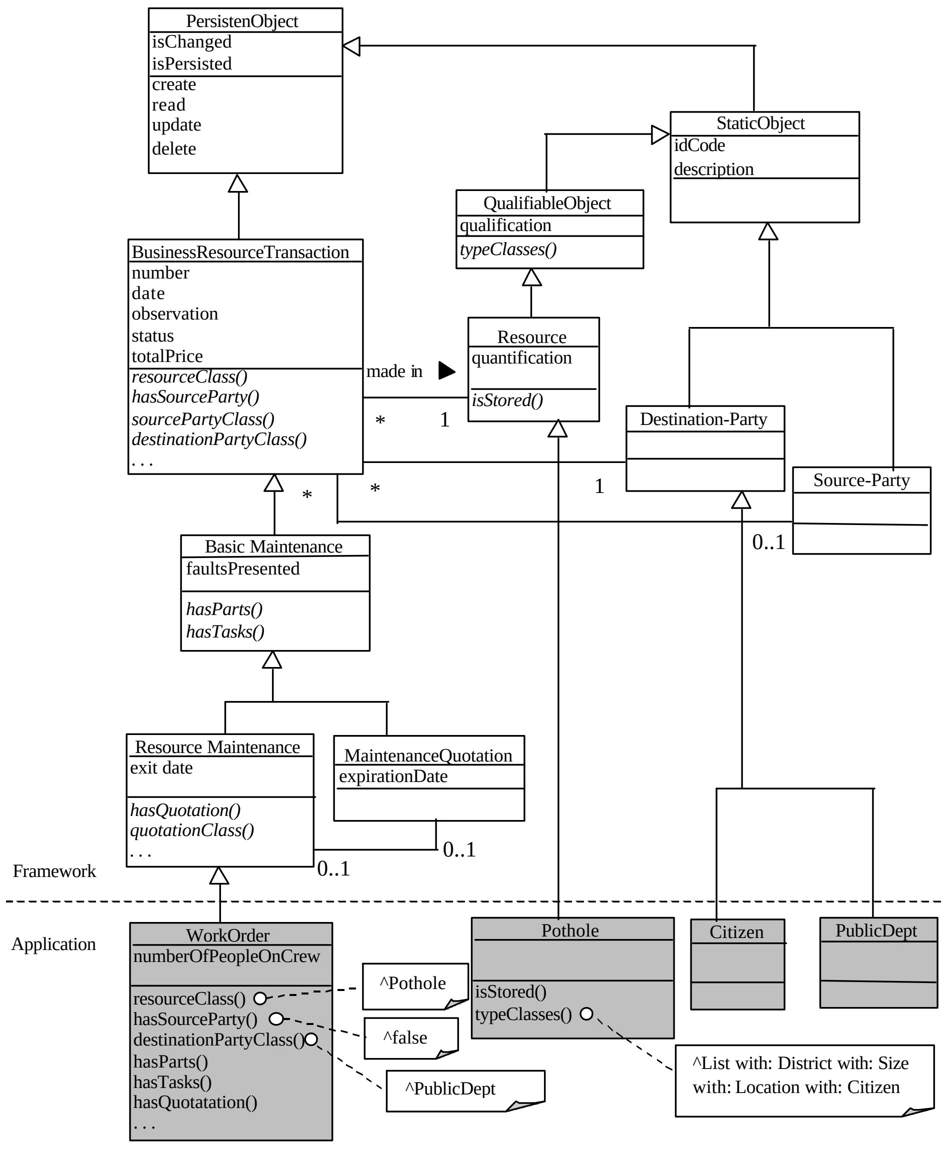


Figure 5. Part of GREN Class Hierarchy and derived classes in the application

appear in the menu are supplied by GREN, for free. Notice that only a few reports were added to the menu, but GREN offers a number of other reports through buttons in the GUI forms. For example, when a Pothole is edited, there is a special button in the GUI form that allows reports to be generated of potholes by address order, by district, by identification code, by location etc.

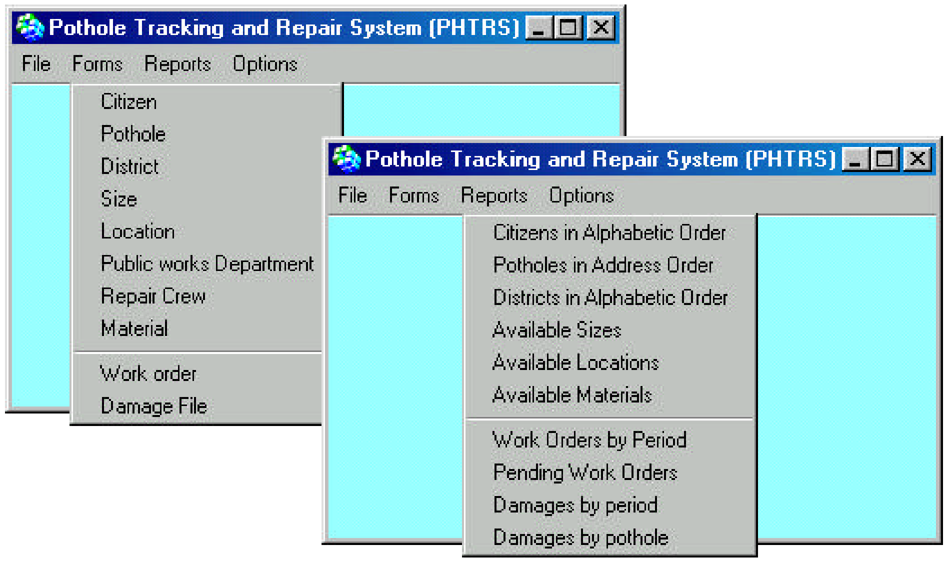


Figure 6. PHTRS Menus

6 Test of the resulting system

In this section we provide some useful information to properly test the resulting application (step 4 of Figure 1). The input for the test is the compiled application code, in the form of classes and methods, the framework documentation, an environment to execute the code, and a test strategy. The output is the tested application code to be deployed.

6.1 General guidelines

The resulting system needs to be tested to ensure both that the requirements have been fulfilled and that the system works in the end user environment. Some observations that need to be taken care of are:

1. Check the framework documentation about the persistence mechanism, in order to know how to create the new database.

2. Follow the instructions of the particular framework to install an executable version of the new system.
3. Use a test strategy to exercise your system and ensure that the requirements have been fulfilled.

6.2 Example

In the case of the GREN framework, the testing step requires the installation of the framework software at a client machine, and the creation of a MySQL database with all the relational tables involved in the patterns used during instantiation. The GREN cookbook has a special section to help its users to create such tables. For the PHTRS example, after creating the MySQL tables, a set of test cases was created to exercise the system. We have executed the menu options of Figure 6 at least for three new objects each and the result has matched the requirements.

We have spent about 9 hours to obtain the final PHTRS system (2 hours with analysis, 1 hour with mapping, 4 hours with implementation, and 2 hours with testing). We have written about 1600 lines of code to adapt GREN to this specific example. These numbers are very low if we consider the number of Function Points (FP) [22,23] of the resulting system, which is about 370 (we have used an adjustment factor of 0.9 to calculate the function points). To obtain the same functionality by programming the applications from scratch would require at least five times more lines of code than it actually did using the framework, considering that Smalltalk requires twenty-one lines of code per function point [24]. Moreover, most of the programmed code consists of methods to deal with new attributes included in the classes, which were not part of the patterns. For example, for each new *Citizen* attribute, there is the corresponding code in the application class to set, get, and initialize it, and there is also code in the user interface class to allow its edition in the input forms. Besides, part of the code is automatically generated by visual programming. This occurs for the GUI forms that were adapted to the specific applications, for example by moving widgets to the appropriate place. Thus, the code that needs to be created is much simpler compared to the application code.

7 Concluding remarks

The approach here proposed intends to ease framework instantiation using pattern languages. Frameworks built using our approach have its architecture influenced by the pattern language, which eases the instantiation of new applications. With the special documentation provided by our approach, framework users basically need 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 for using the framework main functionality. The novel aspect of our proposal is to use the pattern language to allow a smooth transition from analysis to implementation of specific applications. By identifying the patterns that fulfill the requirements of a specific application it is possible to use the framework to implement that application. Our process is general, but a pattern language needs to be developed before the framework construction.

We have used GREN to develop several systems, among which are a car repair management system, a sales system, and a video rental store. We have also conducted a case study with undergraduate students to implement a car rental and a hotel system. GREN attended the functionality of all these systems. With the case study presented in this paper it was also possible to confirm that GREN can be used to model aspects not envisioned before: by examining the syntax aspects of the patterns, we can achieve even more reuse than if we consider only the semantic aspects. This is coherent with Johnson's statement that "a good framework will be used in ways that its designers never conceived" [7]. The system performed well for this flexing usage, avoiding having to program this part from scratch.

Although we have tested our approach only for small applications, with about 10 to 20 application classes each, we believe that it scales well for larger applications. Due to the intrinsic characteristics of patterns, which allow the development of complex systems by partitioning them into smaller units, pattern languages can be extended by adding new patterns to cover more and more aspects of the domain.

We are aware that GRN and GREN cover a restrict domain inside information systems, but our approach can be adapted to allow its usage in a wider variety of domains. Scalability for other computer science areas other than information systems could be an object of future research.

Another important result of our approach is that the framework user knows exactly where to begin the instantiation, as the pattern language guides her/him through the several parts that need to be adapted in the framework hierarchy. This solves a common problem of other approaches mentioned in Section 2, because here the instantiation is focused on the functionality required, with a clear notion of which requirements are attended by each pattern.

References

1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.
2. J. O. Coplien. *Software Design Patterns: Common Questions and Answers*, pages 311–320. Cambridge University Press, January 1998. in L. Rising - The Patterns Handbook: Techniques, Strategies, and Applications.
3. D. Brugali, G. Menga, and A. Aarsten. *A Case Study for Flexible Manufacuring Systems*, pages 85–99. Domain-Specific Application Frameworks: Frameworks Experience by Industry, M. Fayad, R. Johnson, –John Willey and Sons, 2000.
4. M. E. Fayad and R. E. Johnson. *Domain-Specific Application Frameworks: FrameWorks Experience by Industry.* John Wiley & Sons, New York, USA, 2000.
5. R. T. V. Braga and P. C. Masiero. Identification of framework hot spots using pattern languages. In *15th Brazilian Symposium on Software Engineering*, pages 145–160, Rio de Janeiro-Brasil, October 2001.

6. A. Aarsten, D. Brugali, and G. Menga. *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, 2000.
7. R. E. Johnson. Documenting frameworks using patterns. In *OOPSLA '92*, pages 63–76, 1992.
8. D. Brugali and G. Menga. Frameworks and pattern languages: an intriguing relationship. *ACM Computing Surveys*, 32(1):2–7, March 1999.
9. D. Gangopadhyay and S. Mitra. Understanding frameworks by exploration of exemplars. In *International Workshop on C.A.S.E*, IEEE, July 1995.
10. K. Beck and R. Johnson. Patterns generate architectures. In *European Conference on Object-Oriented Programming*, pages 139–149, Bologna, Italy.
11. W. Pree, G. Pomberger, A. Schappert, and P. Sommerlad. Active guidance of framework development. *Software - Concepts and Tools*, 16(3):136–, 1995.
12. A. Ortigosa and M. Campo. Towards agent-oriented assistance for framework instantiation. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2000.
13. R. T. V. Braga and P. C. Masiero. A process for framework construction based on a pattern language, 2002. submitted to the 26th Annual International Computer Software and Applications Conference on February 2002.
14. R. T. V. Braga and P. C. Masiero. Frameworks construction and instantiation using pattern languages, 2002. accepted for publication in the proceedings of the International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications, ACIS, Foz do Iguazu-Brazil.
15. W. Pree. *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, 1999.
16. R. T. V. Braga, F. S. R. Germano, and P. C. Masiero. A pattern language for business resource management. In *6th Pattern Languages of Programs Conference (PLoP'99)*, Monticello – IL, USA, 1999.
17. R. T. V. Braga. GREN: A framework for business resource management. ICMC/USP – Sao Carlos, August 2001. Unpublished, Available on August, 2001 at: `http://www.icmc.sc.usp.br/~rtvb/GRENFramework.html`.
18. Cincom. Visualworks 5i.4 non-commercial, 2001. Available for download on September 25, 2001 at: `http://www.cincom.com`.
19. MySQL. MySQL 3.23 version, 2001. Available for download on September 25, 2001 at: `http://www.mysql.com`.
20. R. S. Pressman. *Software Engineering - A Practitioners Approach, 5th ed.* McGraw Hill, 2001.
21. G. Butler, R. Keller, and H. Mili. A framework for framework documentation. *ACM Computing Surveys*, 32(1), March 2000.
22. A. J. Albrecht. *AD/M Productivity Measurement and Estimate Validation.* IBM Corporate Information Systems, Purchase-NY, USA, 1984.
23. C. Jones. *Applied Software Measurement.* McGraw Hill, New York, USA, 1991.
24. C. Jones. *Preliminary Table of Languages and Levels.* Software Productivity Research Inc., Burlington, Mass., USA, 1989.