Engenharia de Software - SBU
Engenharia: software
FRAMEWORK
visualizaçoS: software

C.N. P.Q. 1.03.03.00-6

Revista de
**Informática**
Teórica e Aplicada

# Automated Recognition of Design Patterns for Framework Understanding

**Marcelo Campo**[†] **and Roberto Tom Price***

[†]Universidad Nacional del Centro de la Pcia. de Buenos Aires
Instituto de Sistemas- Grupo de Objetos y Visualización
San Martín 57, (7000) Tandil, Bs. As., Argentina.

Universidade Federal do Rio Grande do Sul-Instituto de Informática
Capus do Vale, Bloco IV, Predio 43424
Porto alegre-RS-Brasil
e-mail: mcampo@exa.unicen.edu.ar, tomprice@inf.ufrgs.br

## Abstract

System design is one of the most important tasks in the software development cycle, but it is also one of the most complex and time-consuming tasks. Thus, reuse of existing designs becomes very important. Object-oriented frameworks are generic designs for specific application domains that enable the reuse of designs and domain expert experience. In spite of this, frameworks are not simple to reuse because they are difficult to comprehend, mainly due to a lack of good documentation and supporting tools. In this work, an approach to framework comprehension based on the automated recognition and visualization of design patterns is presented. A tool was built to support this approach, by trying to automatically identify and explain the potential patterns existing in a given design. Experimental results and conclusions of tool utilization are also presented.

**Keywords**: Frameworks, design patterns, visualization.

## 1. Design Reuse and Frameworks

System design is one of the most important tasks in the software development cycle. The design defines, essentially, how the system is divided in parts, the interfaces among these parts and how they cooperate in order to implement the required functionality. A good design makes system change and evolution easier by reducing the impact of local change on the rest of the system when a given part has to be adapted to new requirements. System design is, also, one of the most complex tasks of the development cycle. First, design involves the system organization aspects from the point of view of component implementation for a given problem resolution. Second, design involves cognitive aspects of the user interfaces to be provided, specific features from the supporting hardware platforms, as well as specific knowledge about the application domain. With all these aspects in mind, it is reasonable to affirm that the development of a system that provides a good balance among all these aspects is not a task that can be easily achieved by inexperienced designers [DEU 89][JOH 93][GAM 94]. Therefore, *the reuse of existing designs* becomes very important.

With the advent of the object-oriented paradigm at the beginning of the last decade, a new development paradigm appears, where reuse is an integral part of the paradigm. Inheritance, polymorphism and dynamic binding provide the technology that enables the construction of software by specialization, based on the extension of the functionality of existing components. Through the definition of abstract classes, it is possible to reuse, beyond isolated classes, sets of classes designed as a whole for the resolution of the generic functionality of a given application domain. That definitively means application design reuse.

A set of classes that provides a generic solution for a given application domain is denominated **object-oriented framework** [DEU 89][JOH 88]. Essentially, a framework is constituted by a set of classes that implements a domain specific architecture [BEC 94]. Framework classes provide the generic behavior of any application within its domain, leaving the implementation of specific aspects of a given application to be completed by subclasses. This feature represents an important benefit because, once the framework is understood, developers have to focus just on the solution of the specific aspects of the problem being solved, while the overall control structure of the application is inherited from framework classes. In this way, if domain experts design a framework, users of the framework are reusing, implicitly, the experience of these experts.

Through these characteristics, frameworks offer a great potential to increase the productivity and quality in software development. However, starting to use a framework for building specific applications remains a complex task for a user other than the framework designer. Comprehending a framework is, frequently, much harder than comprehending libraries of components that can be reused independently. In the latter case, it is sufficient to understand component external interfaces. On the other hand, using frameworks in order to obtain the maximum benefit from framework reuse, it is necessary to comprehend the internal design of its classes, how these classes collaborate among themselves, as well as the way instances of those classes collaborate at run time [LAJ 94][LAN 95]. Reaching a detailed comprehension of these aspects is, in general, an expensive and time consuming task. This aspect can be considered as the most restrictive factor of the technology.

The inherent complexity of object-oriented program comprehension, widely discussed in recent literature [HEL 90][WIL 92][JOH 92][DEP 93][DEP 94][LAJ 94] [LAN95], can be considered as one of the main causes of the problem of framework comprehension. However, the limitations for representing abstract designs of the current object-oriented design documentation techniques are another important restricting factor. For that reason, several special documentation techniques, such as *contracts* [HEL 90], *patterns* [JOH 92] and *meta-patterns* [PRE 94] were proposed to aid the user in the comprehension of a framework organization. Some of them also describe how a framework can be used to build applications. Certainly, these techniques are very useful for documenting mature and stable frameworks. Nevertheless, not all the frameworks are described using these techniques and, even if they are, the examination of applications or examples developed using the framework is a good starting point to understand the way its classes are instantiated and related among themselves.

In this context, even if documentation is available, tools that analyze the behavior of applications and examples usually provided with a framework, and that visualize the way the classes are organized and related by message passing represent a valuable complement to aid the comprehension of a framework. Particularly, tools that enable the developer to recognize and visualize abstractions of a higher level than the code, such as design patterns [GAM 94] for example, provide an excellent vehicle for understanding the framework design on a higher level of abstraction than simple visualizations based on classes or interactions among objects. A design pattern is the abstraction of a recurring design problem found in different object-oriented designs. Essentially, a design pattern expresses a design intent, suggesting a generic organization of classes and distribution of responsibilities among them that solve the problem in a flexible way. Currently, design patterns are proposed as a way to produce more reusable and adaptable designs. Also, design patterns provide an excellent vehicle for communicating design solutions among designers, and therefore, for helping to understand such designs in terms of standard solutions to common problems.

In the last years, several tools aimed to help with the comprehension of object-oriented software were described in the literature. These approaches are mainly centered either on providing microscopic visions of program behavior for debugging purposes [BRU 93][STA 94][VIO 94], or on providing alternative visualizations of program data [DEP 93][DEP 94]. Even so, except for the work of Lange and Nakamura [LAN 95], little has been reported on tools that use design patterns to help with framework comprehension.

In this work the results of a reverse engineering approach based on recognizing and visualizing potential design patterns existing in a given framework is presented. This approach is based on the Luthier framework [CAM 96][CAM 97] for building tools for application analysis and visualization by means of reflection techniques based on meta-objects [MAE 87]. Luthier provides flexible support for building visualization tools adaptable to different functionalities of analysis, using a hyperdocument manager to organize the information. This hyperdocument representation supports the flexible construction of different visualizations from the analyzed examples, as well as navigation among different visual representations and textual documentation, by means of explicit support for the edition of electronic documentation books. With this support, once one or several applications developed with a given framework are analyzed, a Prolog representation of gathered data is used to recognize potential design patterns that can exist in the framework design. Prolog is used for representing the rules for design pattern recognition, for building the visualizations of potential patterns, and for generating explanations about these patterns and the reasons that suggested their presence in the design.

This paper is organized as follows. The next section discusses the different aspects that make the process of framework understanding difficult. Section 3 briefly describes the main aspects of the visualization tool and the recognition process, and in section 4 experimental results are presented. Section 5 summarizes related works and Section 6 concludes this work.

## 2. Why are Frameworks Difficult to Understand?

In order to be able to adequately specialize abstract classes, as well as to describe the best way the application can be built through the composition of instances of subclasses of those classes, a user needs to comprehend the detailed design of the framework. In complex frameworks, these classes describe patterns of collaboration among instances, through *flexible* design structures, that is, structures that enable the adaptation (sometimes dynamically) of the general behavior provided by the framework.

A framework represents a tradeoff between a general and a flexible solution. A general solution can deal, without changes, with different variants of a given problem. A flexible solution, on the other side, is a solution that, through little changes on its structure, can be adapted to solve those different variants. General solutions are certainly desirable, but most of the times, they present performance problems or they are limited to very restricted domains [PAR 79]. Flexible solutions can be adapted to every particular case, allowing to exploit those aspects that simplify the solution, in terms of performance and functionality. But, in general, very flexible design structures imply very complex designs and, as a consequence, designs harder to understand.

This complexity is more acute in object-oriented programs. Object-oriented programs are, in general, more difficult to understand than traditional procedural systems (with functional architecture). In this case, the structure of the solution description, through the programming language, corresponds directly with the way the system is executed by the computer. Thus, hierarchical representations provide a visual representation very close to the mental model a programmer has about an executing program. This correspondence simplifies the identification of source portions that produce a given effect in the system execution[1]. Once the functional model has been recovered, which may not be a simple task, it is relatively easy to reason about the system behavior and determine which portion of code should be modified. In object-oriented programs, on the other hand, the relationship between source code structure and the execution model may not be so direct, whence the consideration of two perspectives: the dynamic one and the constructive one is necessary [BUH 92].

Lange and Nakamura point out, as another problem related to framework comprehension, the difficulty that developers find when trying to document them [LAN 95]. Framework documentation presents more complex problems than conventional system documentation, either object-oriented or not. This complexity can be attributed, essentially, to two main reasons: the abstract nature of the design and the goal of any framework, that is, its reuse. On one hand, documentation has to show how to use the framework for application building, without giving details about internal behavior. On the other hand, detailed framework design description is also needed. While documentation of use is useful for occasional users, framework documentation helps to use the framework in applications other than the ones previewed by the original designer [JOH 92]. A clear comprehension about the way a framework works is a great aid for using all its potential.

## 2.1 The Object Model

The typical structure of an executing object-oriented program is a graph, where nodes represent objects and links represent object references. Objects are dynamically created and destroyed, causing the topology of the graph to change dynamically. In spite of this, object-oriented programs are not built by configuring objects and object references, but by the use of concepts such as classes, polymorphism and inheritance. These building mechanisms allow to divide a system in reusable components, which are implemented in run time in the simpler instance network model. This dichotomy between both models is one of the aspects that make object-oriented programs harder to understand than conventional programs. That is, in order to comprehend an object-oriented application it is necessary to comprehend the dynamic and constructive visions, as well as the relationships established among them. This implies the building of the reverse mapping, from the simple instance network model to the hierarchy class model that describes it.

Polymorphism and dynamic binding are the essential mechanisms that enable the construction of flexible software and, as a consequence, frameworks. Besides, they are two essential factors that contribute to the difficulty in comprehending object-oriented programs. Polymorphism, dynamic binding and inheritance allow very complex behavior inside the same class hierarchy, which can only be completely understood at run time. Examples of this kind of behavior are the classes designed to dynamically add functionality to objects, called *wrappers* [GAM 94]. Most of the wrapper behavior is to propagate the messages it receives to its component. This produces the execution of different

1 This relationship is valid for sequential programs.

redefinitions of a same method, in different levels of the same hierarchy. The real method to be executed depends on the configuration of instances at each execution point. This behavior, known as *yo-yo* effect [TAE 89], makes it hard to statically determine the method to be executed in a given invocation. As a consequence, it is necessary to analyze the code of multiple classes, until the relationships among different hierarchy components are comprehended.

## 2.2 The Role of Design Patterns

A framework is, essentially, the implementation of a generic architecture for an application domain in terms of classes. A previous knowledge about the application domain is, doubtless, of great importance to help with a given framework comprehension. Through the general domain knowledge, a programmer is able to comprehend the general organization of concepts or, more specifically, the domain model implemented by the framework. On the other hand, it is also necessary to take into account that the goal of a framework development is to allow framework users to reuse the designer domain knowledge. Therefore, it is reasonable to expect that the framework users do not have a deep knowledge about the application domain, but just the essential knowledge about the functionality of the application to be implemented. Ideally, in order to be actually useful, a framework should allow the user to implement applications knowing just the functionality that abstract classes leave to be implemented by subclasses. Therefore, a reasonable first step in a framework comprehension process is to provide the user with the mechanisms that allow her to build an initial mental model of the structure and the behavior of the architecture implemented by the framework. According to this, providing support for recognizing abstractions not supported by the programming language, is an important complement to facilitate the global comprehension of the functionality of a framework.

Design patterns [GAM 94] represent design abstractions that are not supported by current object-oriented languages, but are of great importance in comprehending how system objects are organized and collaborate in order to satisfy the global functionality. A design pattern names a given combination of classes and methods that solve a general, recurring, design problem. For example, the *Composite* design pattern (Fig. 1) prescribes how to organize objects that are recursively composed to form a hierarchy of parts. The pattern enables a uniform treatment of single and composed objects through the definition of a common interface to the services provided by single and composite objects. Single objects directly manage requirements from external clients, while composites propagate requirements to their components.

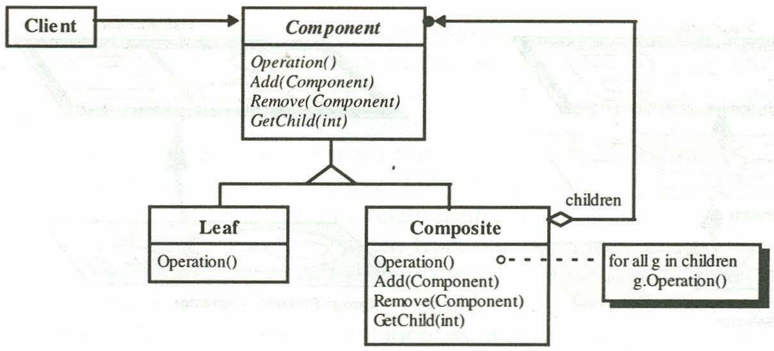


Fig. 1 Class Structure of the *Composite* Design Pattern

If a user knows the problem that a given pattern is intended to solve, and what classes and methods the pattern prescribes for the generic solution, then this user can quickly understand the nature of the relationship established among framework classes without a very detailed analysis. In this way, the identification of potential design patterns that can exist inside a framework structure is an important complement to improve the comprehension of how determined parts of the framework were designed and the function that some methods have inside a given class.

It is necessary to take into account that an approach exclusively centered on recognizing and visualizing design patterns is not enough to completely guide the process of framework comprehension. Design patterns can be useful to drive the process of framework design at architectural level, but not all the framework structures can be derived from the design patterns described in, for example, the catalog presented by Gamma et al [GAM 94]. The number of patterns in the catalog, which are the most widely known, is relatively small and they vary a lot in their level of abstractions and the domains where they are useful. Therefore, recognizing these functional units in a framework becomes an important complement in order to provide the user with more abstract initial visions of the framework structure.

## 3. Looking for Design Patterns with Luthier

The Luthier framework was designed and implemented in Smalltalk, with the goal of providing a flexible support for the construction of tools for object-oriented framework analysis and visualization, through reflective techniques based on meta-objects [MAE 88]. A distinctive characteristic of Luthier is the sub-framework for meta-object support based on the concept of meta-object managers [CAM 96]. Through this support customized meta-object protocols, specially adapted for different dynamic program analysis functions, can be implemented with little effort. Specific meta-object classes can be implemented to extract relevant static and dynamic information from the analyzed program, and to build an abstract representation of the framework. With this information, different abstractions, such

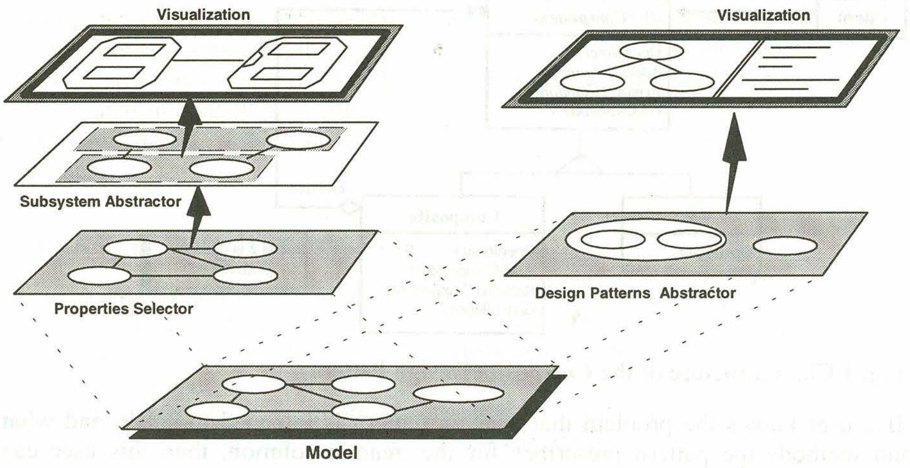


Fig. 2 Relationship among visualizations, abstractors and information representation

as subsystems and design patterns, can be deduced, and different visualizations can be built using the visualization sub-framework.

Luthier introduces the concept of *abstractor objects*, which explicitly separate the information representation from the visualizations (Fig. 2). Abstractors represent a generic architectural component of tools, where different analysis algorithms and selection criteria can be located, without the need of modify either the classes of the information representation or the classes implementing visualizations. They also provide a generic support for symbolic abstraction scales, which enable the semantic zoom of visualizations without the need to program special filters in visualizations. An abstraction scale is an ordered tuple naming the order in which constructions, such as subsystems, classes, methods, and so on, should be visualized. A scale has its own user-interface control (usually a slider) through which the user can interactively vary the level of abstraction of the visualization (i.e. dynamically showing or hiding details). The visualizations, in turn, only have to worry about what must be shown in the current abstraction level, according to the data that abstractors pass to them.

This powerful feature enables the combination and reuse of different algorithms for abstraction recognition with different visualization styles, as for example, subsystem analysis, structural relationship analysis and design patterns.

## 3.1 Recognizing Design Patterns

Recognizing potential design patterns that may exist in the structure of an analyzed framework is, essentially, a *pattern matching* problem. Pattern matching techniques are used by most of reverse engineering tools, but in general, they are limited to small scale problems. Graph-based representations have the limitation of the impossibility for checking the existence of a given pattern among other patterns intermixed in the program code [FIS 91]. In the case of object-oriented programs, recognizing structural patterns through pattern matching presents the same limitation, complicated by the dynamic nature of the graph defined by the configuration of the instances. Prolog-based representations, on the other hand, can simplify the process of pattern matching, by look for abstract properties of the control flow rather than only its structure. Different patterns can be expressed in terms of rules that define the static and dynamic relationships that characterize each pattern, as well as, heuristics that help to identify a pattern from other similar patterns. This last aspect is particularly relevant in the case of design patterns because several patterns have a very similar static class structure, it being only possible to distinguish one from another by its dynamic behavior, having no type information, as in Smalltalk. Besides, the materialization of design patterns can depend on the implementation language and the particular requirements of each application. Therefore, it is not possible to guarantee that all the patterns that could exist in a framework can be automatically recognized through a pattern matching process.

Also, it is necessary to take into account that a design pattern expresses a design intent, suggesting a generic class structure to organize and distribute responsibilities among them, which is almost identical in many cases. Evident examples of this fact are *Composite*, *Decorator* and *Proxy* design patterns. *Composite* and *Decorator* have a very similar class structure, but their design intents are different. While *Decorator* adds functionality to the decorated object, *Composite* has the goal of object composition. From the dynamic behavior point of view, both patterns are characterized by propagating messages to its components. The only difference is that *Decorator* is composed of just one component, while *Composite* often has more than one. *Decorator* and *Proxy* have an identical class structure, differing in the design intent. The goal of a *Proxy* is to control the access to the object that it represents. In this case, the distinctive behavior among them, is that generally case, a *Decorator* always propagates messages to its component, while a *Proxy* might conditionally propagate such messages. Similar problems also arise with *State* and *Strategy*.

The strategy adopted for pattern recognition takes advantage of the following Smalltalk characteristics:

- Decorator and Proxy patterns are recognized if they were implemented using the doesNotUnderstand: method, which is the most common way to delegate responsibilities among objects. If an object receives a message not implemented by it, the doesNotUnderstand: method is automatically invoked by the run-time system. The Decorator and Proxy patterns are usually implemented by redefining this method for passing a request to its component.
- Pattern Observer has a Subject object, which updates all its dependent objects, called observers, every time its internal state changes. This mechanism is already implemented in Smalltalk, in the Object class, that provides the behavior need to manage the object dependent list, and to inform every dependent object the changes on the observed object. When this object changes its state, it announces the change by sending itself the changed message that makes dependent objects receive the update message.

- In the representation of the collected information, methods are classified according to their categories. Thus, recognition of TemplateMethod pattern is trivial.

## 3.2 Prolog-based Representation

An abstractor that provides pattern information to the visualization encapsulates the generic pattern recognition mechanism. Hypertext model information representation is converted to Prolog format, according to the following convention:

```
class (ClassName, Superclass, RootClass )
message (OriginMethod, OriginClass,OriginRootClass, OriginMethodCategory,
         OriginMethodType, TargetMethod, TargetClass,TargetRootClass,
         TargetMethodCategory, TargetMethodType)
```

This terms represent class and message information needed for recognition. The last term represents communication among components (control flow), as well as communication inside a component. For both types of methods, the following information is stored:

- method name
- method implementing class
- method category (abstract, hook, template or base)
- method type (instance or class)

For example, if OriginClass and TargetClass are the same, communication is occurring inside the component, otherwise the communication is occurring between different components. If OriginMethod and TargetMethod are the same, and communication is occurring inside a component, it is considered a "recursive" invocation; but, if communication is held between different components, it is considered message propagation.

Individual pattern characteristics are codified by Prolog rules that search through the data base message and class combinations that satisfy each particular rule. There are general aspects for all the patterns belonging to the same category, and with the addition of every pattern's specific aspects, the rule is defined. For example, structural patterns are characterized by the way objects are composed in order to achieve more functionality. Therefore, the rule for recognizing this kind of patterns is based on finding composed objects. The specific rule for *Composite* pattern detects that a Composite object, when it receives a message, propagates it to, at least, two of its components. The Composite and its components should have a common Superclass.

```
% Checks the existence of at least two subclasses to which messages are
%propagated from a superclass
Composite():-
```

```
1) message( Operation, Composite, FatherComposite, _, _, Operation,
   Component1,                   FatherComponet1, _, _ ),
2) message( Operation, Composite, FatherComposite, _, _, Operation,
   Component2,                   FatherComponet2, _, _ ),
3) isReceiverVarInst( Operation,Operation, Composite ),
4) hasCommonSuperClass( Composite, Component1 ),
5) hasCommonSuperClass( Composite, Component2 ),
6) assert( composite( Composite, Operation,[ [Component1, Operation],
                                [Component2, Operation] ] ) ),
7) fail.
```

**Composite():- !.**

The terms of the rule numbered **(1)**, **(2)** and **(3)** verify if the *Composite* class propagates the request *Operation* to, at least, two components: *Component1* and *Component2*. The *hasCommonSuperClass* predicate, used in **(4)** and **(5)** determines if classes *Composite*, *Component1* and *Component2* have a common superclass.

Object composition in Smalltalk is implemented as a reference to an object, by means of an instance variable. The auxiliary predicate *isReceiverVarInst(M1, M2, C)* analyzes Smalltalk code and verifies if the **M2** method from the **C** class invokes the **M1** method, where the receiver of method **M1** is an instance variable of **C**[2].

Another example is the *Strategy* pattern. The pattern intent is to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it [GAM 94]. The rule for Strategy pattern recognition searches for a class defining a common interface for all supported algorithms (an abstract method), and verifies if this method is redefined in all its subclasses. This strategy method must be invoked by a client through a reference held by the client (instance variable), and this client should not be on the same inheritance hierarchy of the Strategy.

*%Cheks for a class defining an abstract algorithm.*
**Strategy():-**

```
1) message( ClientMethod, ClientClass, _, _, _, StrategyMethod,
   ConcreteStrategy,
               Strategy,abstract,instance),
2) class( ConcreteStrategy, _, Strategy ),
3) allSubclassesRedefined( Strategy, StrategyMethod,_ ),
```

[2]This checking verifies if the receptor object is referenced directly or through a collection. This information is gathered by Luthier during the analysis phase and maintained in the hipertext representation of the class.

```
4) isReceiverVarInst( ClientClass, ClientMethod, StrategyMethod),
5) not( relatedClasses( ClientClass,Strategy ) ),
6) assert( strategy( Strategy, StrategyMethod, [ClientMethod, ClientClass,
                                         StrategyMethod, ConcreteStrategy ] ) ),
7) fail.
```

**Strategy():-** !.

Terms **(1)** to **(3)** state that Strategy class defines the interface of the method and that every subclass redefines the method. Terms **(4)** and **(5)** verify the conditions that have to be fulfilled by the client. Term **(6)** stores the potential pattern.

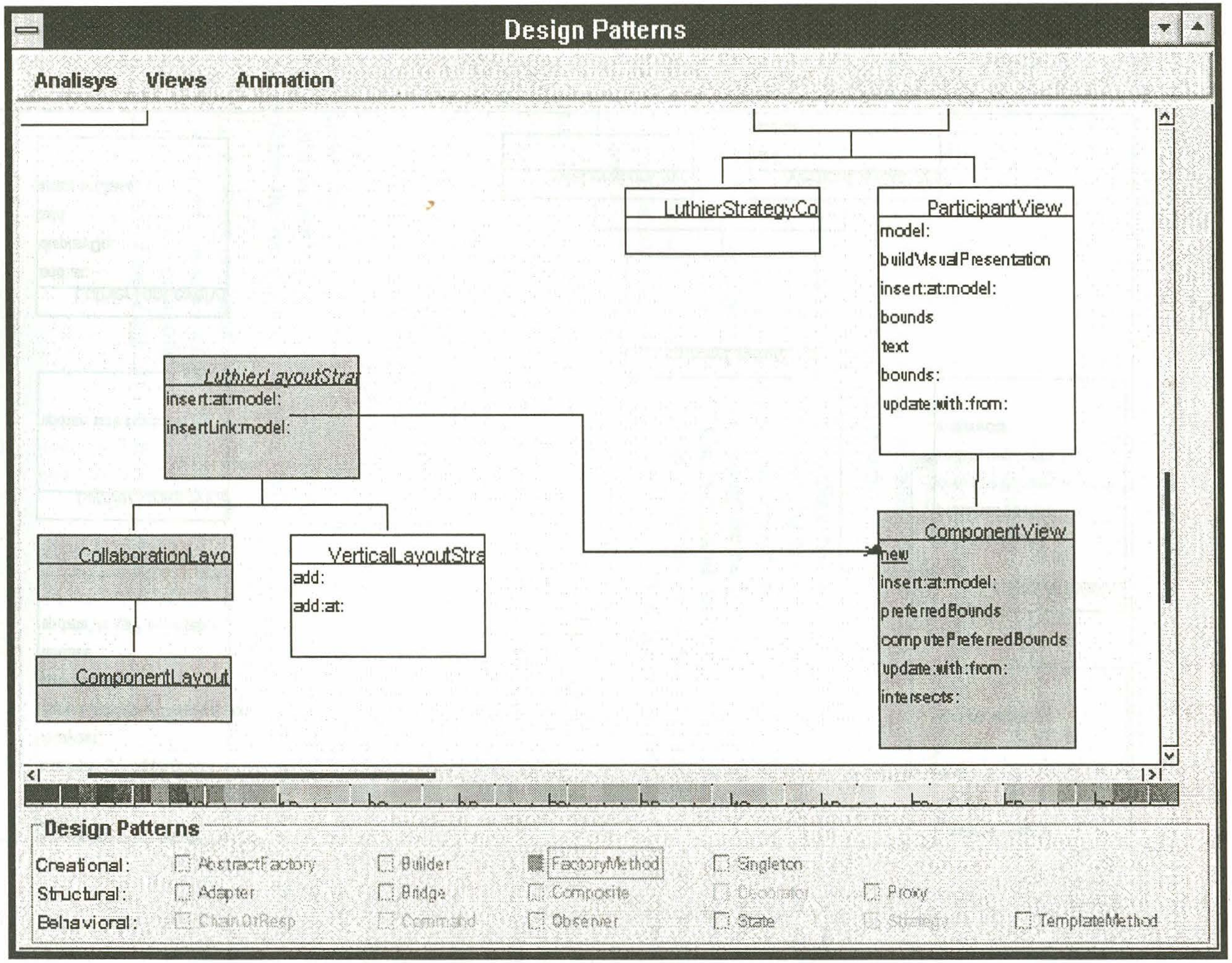
The rules shown above were simplified for clarity's sake. The actual rules are more complex, as many details have to be considered for a more accurate pattern recognition. Even so, the rules described are sufficient so as to give a good idea about their general structure. The rules for the rest of the design patterns that can be recognized respond to a similar structure.

## 3.3 Visualization

Fig. 3 presents a snapshot of the graphical browser provided by Luthier to visualize design patterns recognized in the structure of an analyzed framework. In this browser, an extended OMT notation is used to indicate the messages and methods involved in each pattern and colors (not shown in figure 3) are used to enhance comprehension. The lower pane presents the complete list of design pattern names, highlighting with different colors those patterns that were recognized during the analysis phase. The selection of a pattern from the list, highlights in the visualization the classes involved in that pattern with its corresponding color. This enables the independent analysis of each pattern, as well as, the navigation to the alternative message flow visualization, in order to analyze the dynamic behavior of the pattern. Alternatively, it is possible to highlight with the corresponding color those methods and messages that define each selected pattern. The first visualization is helpful to focus the user attention on occurrences of particular patterns, whereas the second alternative is useful to visualize those patterns that define the design of a given class.

The example of Fig. 4 shows the visualization of the *FactoryMethod* design pattern in the class *LuthierLayoutStrategy.* This pattern is suggested because the *insert:at:model:* abstract method is redefined in *CollaborationLayoutStrategy* and *ComponentLayoutStrategy* subclasses, which creates objects of type *ComponentView,* belonging to a different class hierarchy.

Similarly, the Luthier framework [CAM 97] identifies, draws and explains other existing design patterns.



Design Patterns

Creational: ☐ AbstractFactory ☐ Builder [✓ FactoryMethod] ☐ Singleton

Structural: ☐ Adapter ☐ Bridge ☐ Composite ☐ Decorator ☐ Proxy

Behavioral: ☐ ChainOfResp ☐ Command ☐ Observer ☐ State ☐ Strategy ☐ TemplateMethod

Fig. 3. Visualization of the *Factory Method* Design Pattern

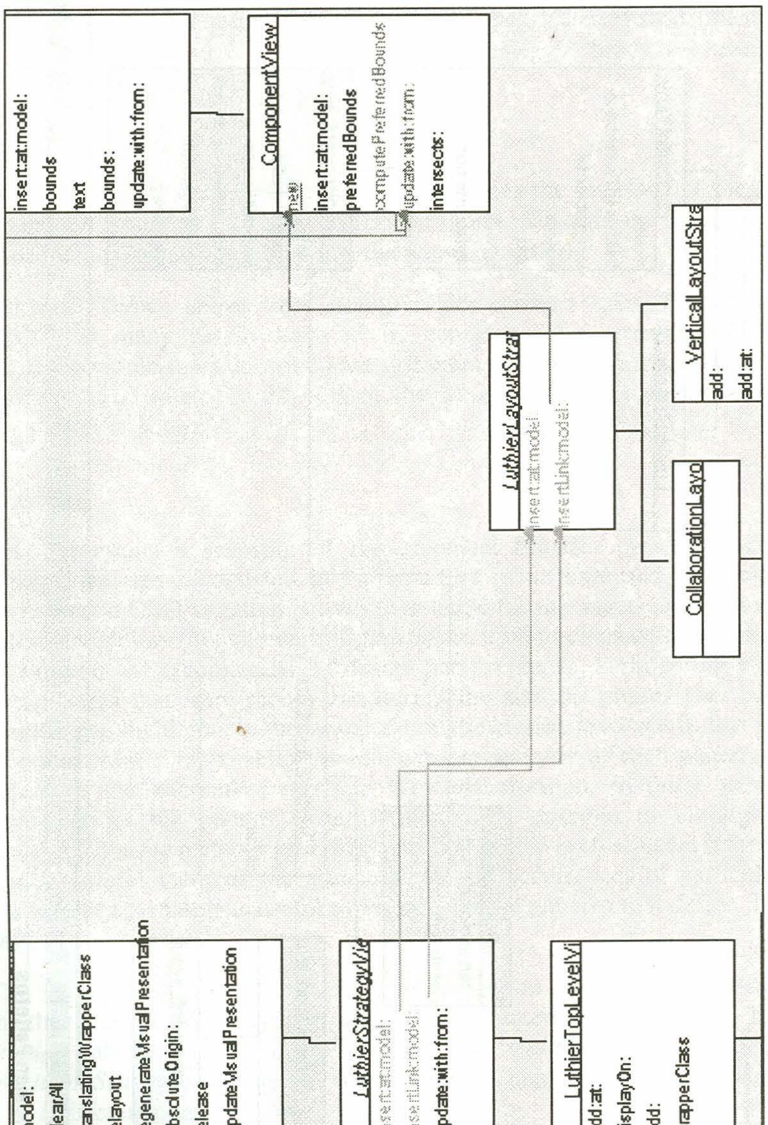


Fig. 4 Visualization of *Strategy* and *Factory Method* in the *LuthierLayoutStrategy* class

### 3.4 Explaining Recognized Patterns

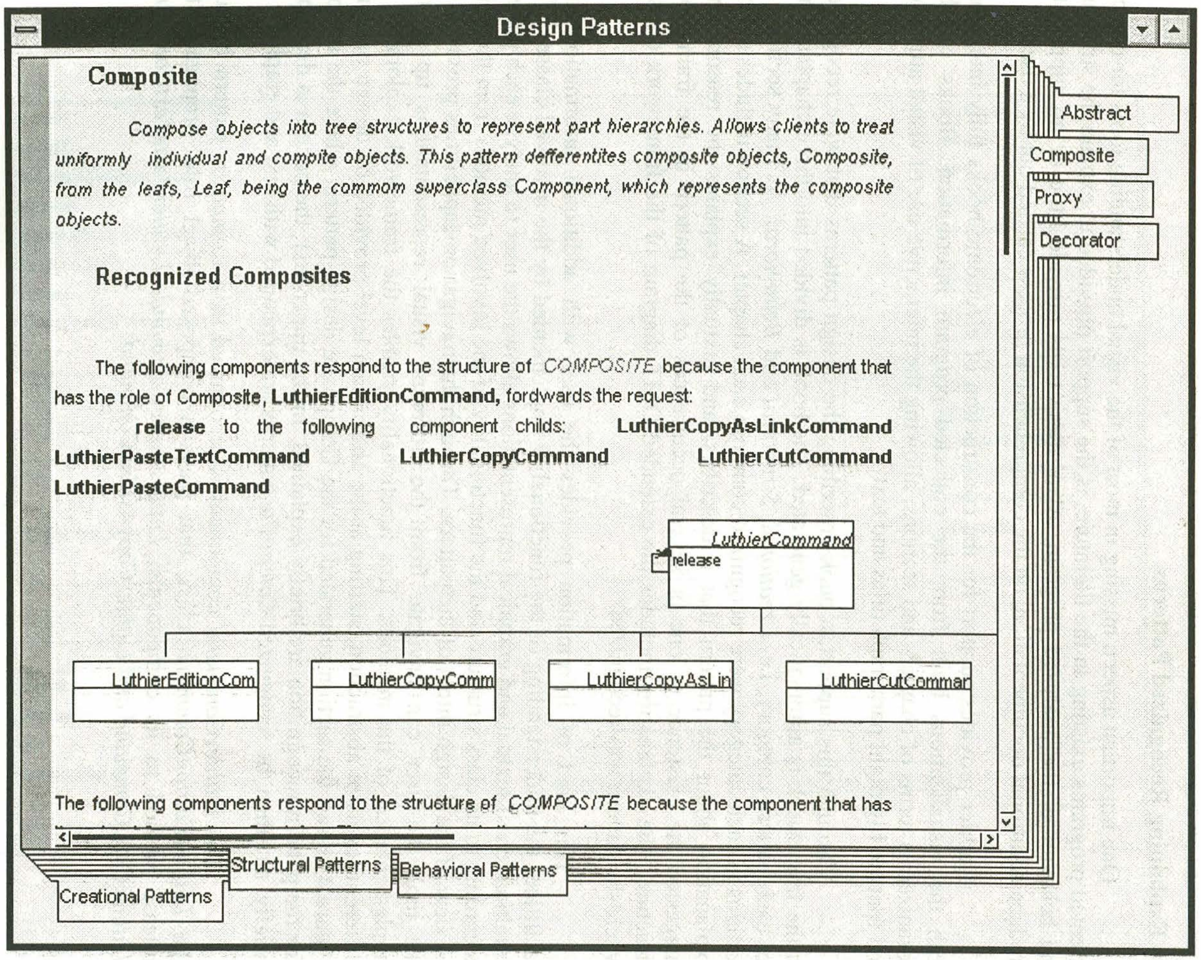
One important aspect, missing in most of the visual understanding tools for object-oriented programs existing in the literature, is the support provided to explain the structure and behavior of the analyzed program. Also, these tools provide little or no support to produce additional documentation, on the comprehension process carried out by a user.

Luthier provides support for the construction of electronic books fully integrated with the visualizations built from the collected program information. Books can be organized in terms of chapters and sections, allowing the interactive use of visual attributes and fonts to highlight paragraph titles and text.

Through this support, a book describing the design patterns that were recognized in the framework is automatically generated. The book is divided into three chapters, one for each pattern category, i.e. *Creational*, *Structural* and *Behavioral.* As many sections as patterns in each category were recognized compose each chapter. A section includes a short explanation about the pattern that it describes, and textually explains the reasons that suggested the existence of one or several occurrences of the pattern in the framework structure. The explanation includes the extended OMT diagram of the framework classes where the pattern was recognized.

This kind of information provides the user with additional information that facilitates the understanding of the functionality implemented by the involved classes. The combination of textual and graphical representations allows the user to analyze each pattern according to its class structure and the functionality of the involved methods from the point of view of the design intent of the pattern. Through the navigation capabilities provided by the tool, the user can navigate from the different visual representations up to the implementation of the methods. This functionality enables the framework exploration at different levels of abstraction, starting at the abstraction level provided by design patterns. Figure 4 shows the section generated for the Composite design pattern, where the reasons for the pattern recognition are briefly explained. In the upper text, the pattern is described briefly, and then the recognized *Composite* patterns are explained with text and graphics.

The *LuthierCommand* component is recognized as a *Composite* pattern because the *LuthierEditionCommand* class, that plays a composed-object role, propagates the release method to its components: *LuthierPasteCommand, LuthierPasteTextCommand, LuthierCopyCommand and LuthierCopyAsLinkCommand.*



**Composite**

*Compose objects into tree structures to represent part hierarchies. Allows clients to treat uniformly individual and compite objects. This pattern defferentites composite objects, Composite, from the leafs, Leaf, being the commom superclass Component, which represents the composite objects.*

**Recognized Composites**

The following components respond to the structure of *COMPOSITE* because the component that has the role of Composite, **LuthierEditionCommand,** fordwards the request:
**release** to the following component childs: **LuthierCopyAsLinkCommand LuthierPasteTextCommand LuthierCopyCommand LuthierCutCommand LuthierPasteCommand**

The following components respond to the structure of *COMPOSITE* because the component that has

Fig. 5 - Explanation of Composite Patterns Recognized in Luthier

## 4. Experimental Results

In order to test the effectiveness of the approach and the tool, an experiment on framework utilization for building applications was carried out. The experiment does not precisely demonstrate the efficiency level of neither the framework comprehension approach nor the tools built to support it, but it gives a good idea about their advantages as well as their limitations. It consisted in building a Petri Net editor with typical editing functionality, using the well-known framework *HotDraw* [JOH 92], for construction of graphical editors.

The most important aspects of the editor's design are the definition of figures to be edited and the utilization of the constraint system. because the rest of the functionality is almost completely implemented by the framework, and it is simple to be copied from existing examples. In this way, through the experiment is was possible to test the framework comprehension, on class redefinition aspects as well as utilization of existing classes.

Three groups of students of the System Engineering course were used for the experiment. Two of them used the set of tools developed with Luthier for supporting framework comprehension (groups GL1 and GL2), while the third group (GN) did not use the tool. Besides, group GL2 used the complete set of tools, including the design pattern recognition and visualization but group GL1 did not use the functionality related to patterns.

The resultant applications were compared using metrics based on [LOR 94]. The results obtained from the metrics can be interpreted in different ways, depending heavily on the type of the analyzed program. When comparing the level of framework reuse, most of the metrics do not provide information that can be considered very significant, although they offer interesting suggestions about the relatives differences among the three applications, which can be verified through further analysis. The metrics highlighted in Table 1, are those that, on average, are particularly interesting as a suggestion about the framework reuse achieved by each application, as well as the design quality of an application compared with the others. The complete result analysis can be found in [CAM 97].

The specialization index of groups GL1 and GL2 are nearly equivalent, while the index of the GN is significantly lower. This index shows a low level of method redefinition and reuse of the functionality provided by the framework, as the number of new methods is high. On average, though, GL1 presents the greatest number of new methods per class and the highest nesting level. In this case, the specialization index is complemented for a great number of inherited methods. The relationship between redefined and added methods shows a parity between groups GL1 and GN, while the value for GL2 is two times greater. The combination of both index suggests a better degree of reuse for group GL2. The other metrics do not provide significant information, excepting the following aspects:

- *The great number of class variables defined by GN, which are completely unnecessary.*

- *GL2 presents the lower number of classes without instances and uncalled methods, suggesting a better use of the available functionality.*
- In order to determine which is the best solution between GL1 and GL2, a detailed analysis of the applications was made. Here, similar information from that provided by the metrics can be extracted: the application developed by group GN presents the greater problems related with the editor design, while the other two applications present little difference between them.

'Comparing the results in a general way, the main difference between the design decisions of the three applications is related to the design of the figures to be edited and the utilization of the constraints:

- GL1 presents a better utilization of the constraint system, which allows them to easily solve some problems that arose due to bad decisions about class specialization.
- GL2 presents the better structure, in terms of reuse of the framework functionality, due to an adequate selection of the classes to be specialized, but they make a weak utilization of the constraint system.
- GN presents problems on the reuse of behavior implemented by the framework, on the aspects related with abstraction designs as well as on the use of the constraint system.

From the analysis of the time used by each group [CAM 97], it can be seen that GL1 had more time dedicated to comprehension activities and less time to development, while GL2 was the group that used less total time. The difference between the times of tool utilization

| **Metric** | **GL1** | **GL2** | **GN** |
|---|---|---|---|
| **Totals** | | | |
| **Number of Classes** | **16** | **12** | **32** |
| Lines of Code | 1427 | 980 | 1854 |
| **Nesting Level (max.)** | 7 | 7 | 8 |
| **Number of Methods** | **178** | **148** | **281** |
| Number of Inherited Methods | 2040 | 1569 | 4082 |
| Number of Redefined Methods | 102 | 95 | 126 |
| **Number of Added Methods** | **76** | **53** | **155** |
| **Number of Messages** | **815** | **781** | **1370** |
| Number of Class Variables | **0** | **0** | **16** |
| **Classes w/o Instances** | **9** | **4** | **18** |
| **Number of Methods not Called** | **51** | **25** | **115** |
| Number of Public Methods | 94 | 75 | 125 |
| **Averages** | | | |
| Number of Inherited Methods / Class | 127.50 | **130.75** | 127.56 |
| **Number of Redefined M. / Class** | **6.37** | **7.91** | 3.93 |
| **Number of Added Methods / Class** | 8.56 | **4.41** | 6.85 |
| **R/A Proportion** | 0.63 | **1.55** | 0.61 |
| **Specialization Index** | **3.47** | **3.20** | **2.36** |

Table 1- Metric values

is the reason for the much better use of the constraint system by group GL1.

Some important conclusions can be extracted from the experiment described above about the utilization of the framework comprehension tool, but they are not definitive because of the small size of the sampling it represents:

- The tool helped to obtain much better results, in terms of the reuse of the functionality provided by the used framework and quality of the final work.
- The tool can induce an exploration of details that are not necessarily relevant. Nevertheless, the comprehension of these details helped the group GL1 to use a very complex subsystem, such as the constraint system, very well.
- The tool does not necessarily help to·reduce the development time.
- The total time used by GL2 group shows that design patterns induced more adequate design decisions, making the framework exploration easier.

## 5. Related Work

The use of visualization and animation techniques to assist object-oriented program understanding is being specially explored in the area of program debugging. Most of the current systems are based on event generation mechanisms. Events are used to inform the visualization system on, for example, the sending of messages, instance creation/destruction and method entry/exit. Event-based mechanisms are especially suitable for program animation tools because they support the definition of events at any level of abstraction.

The BEE++ application framework [BRU 93], provides a platform to build tools for dynamic analysis of distributed programs. It supports event monitoring, visualization and graphic debugging-tools distributed across different nodes of the network. Luthier does not support the analysis of distributed frameworks, but the use of meta-objects could enable the transparent monitoring of such applications too.

VizBug++ [STA 94] is a visualization system for C++ programs that integrates diagrammatic views of the class hierarchy, instances and flow of control. The system is based on events to produce smooth animation of the program execution through arrows that represent method and function invocations. Vion-Dury and Santana [VIO 94] addressed the use of 3D visualizations. They introduced the concept of virtual images for debugging distributed object-oriented applications. A virtual image is a graphic representation of an object that uses a 3D spatial model. Objects are represented by polyhedrons that have significant shapes, colors, volumes and orientation. From a cognitive point of view, this representation offers interesting possibilities to represent more abstract structures. However, it does not seem certain that text can be entirely substituted by polyhedral shapes.

DePaw, Helm, Kimelman and Vlissides [DEP 93,94] proposed matrix-based visualizations of the dynamic behavior of C++ programs. They use multiple views to represent different aspects of execution data, using colors to denote instance creation/destruction frequency, *inter* and *intra* class invocations, instance-allocation history, among others. These representations are generated through a portable platform for instrumenting C++ classes, enabling the generation of interesting events and the control of the program execution. These representations do visualize partial aspects of program behavior and support navigation functions, but it does not emphasize aspects concerning frameworks as those discussed in this paper.

*Software Refactory* [OPD 92] is the first example of using reverse engineering tools to support framework development. This tool supports the restructuring process of a framework programmed in C++, starting from the static analysis of applications built with that framework. *Software Refactory* implements in a semi-automatic way several class refactorizations, as for example: creating abstract superclasses from concrete classes, division of a class into different subclasses, and substitution of inheritance relationships by aggregation relationships, among the most important. These processes imply the class hierarchy reorganization, moving methods between classes and the creation of new methods. Nevertheless, the changes that have to be done to the code are simple, because the common behavior can contain aspects specific to each implementation. The semantic of several constructions can not be inferred through lexical and structural analysis and, therefore, it is necessary for the designer to determine when a given transformation must be applied or not. Even so, this feature is desirable, because the refactoring process is an activity to be carried out by the framework designer, who knows the reasons why some design constructions were implemented in specific ways. Software Refactory provides a valuable support for code manipulation and restructuring, but it does not provide any support for documenting the result of factorizations that were made.

The work described in this paper is heavily related to the work of Lange and Nakamura on the Program Explorer [LAN 95]. They also propose the use of interactive program visualization based on design patterns as the way to obtain structured access to the interaction of framework components. Their work intends to provide a uniform Prolog-based model to represent static as well as dynamic framework information, but it does not explicit how design patterns are automatically recognized, or even if actually they are.

## 6. Conclusions

An approach to framework comprehension and a tool to support were presented. Results of experiments carried out with the tool strongly suggest that the use of the tool lead to better design decisions and a deeper knowledge about the analyzed framework. On the other hand, development time seems to remain stable, independently of the tool utilization. That is, the tool neither reduces the time needed to develop an application nor imposes a delay on this development. However, the use of the pattern visualization would enable to reduce these times, by making the framework exploration easier.

Current results are promising, but further and more comprehensive tests should be done, in order to get more information about the effect of the tool utilization on framework-based application development. These tests will allow to enhance the framework comprehension approach and the supporting tool.

Regarding the design pattern recognition, through the current rules it is possible to recognize patterns in cases where their existence is relatively evident. Even so, the rules can be extended and improved to recognize implementations that are not so evident, adding static information as, for example, method code structure. With this information, it could be possible to try to determine with higher precision the existence of patterns not recognized with the current rules.

The recognition of design patterns is a very time-consuming task due, essentially, to the inherent inefficiency of the Prolog interpreter, which is a program written in Smalltalk itself. This double interpretation level produces a relatively poor performance. Besides, the rule implementation is extremely inefficient, being another source of poor performance of the functionality. This inefficiency, however, can be tolerated if the useful information provided by the tool is considered.

## References

[AMA 97] Amandi, A; Price, A. *Towards Object-Oriented Agent Programming: The Brainstorm Meta-Level Architecture*. Procs. of 1st Autonomous Agents Conference, Los Angeles, ACM Press, February 1997.

[BEC 94] Beck, K.; Johnson, R. *Patterns Generate Architectures*. Procs. ECOOP'94, Bologna, Italy, Berlin:Springer-Verlag, 1994. p. 89-110.

[BIG 87] Biggerstaff, T.; Richter C. *Reusability Framework, Assessment, and Directions*. IEEE Software, March 1987.

[BIG 93] Biggerstaf, T.; Bharat, G.; Webster D. *The Concept Assignment Problem in Program Understanding*. Procs. 15th. IEEE Intl. Conf. on Software Engineering, Baltimore. Los Alamitos: IEEE Press, 1993.

[BRO 83] Brooks, R. *Towards a Theory of the Comprehension of Computer Programs*. International Journal of Man-Machine Studies, v.18, n.6, p. 543-554, June 1983.

[BRO 84] Brown, M. Sedgewick, Robert. *A system for Algorithm Animation*. ACM Computer Graphics, v.11, n 7, July 1984, p. 177-186

[BRO 88] Brown, M. *Exploring Algorithms using Balsa-II*. IEEE Computer, v.19, n.5, p 14-36, May 1988.

[BRU 93] Bruegge, B.; Gottschalk, T.; Luo, B. *A Framework for Dynamic Program Analyzers*, Procs. OOPSLA'93, Washington D.C. New York:ACM Press, Oct. 1993.

[BUH 92] Buhr, R.; Casselman, R. *Architectures with Pictures*. Procs. OOPSLA'92, Vancouver, Canadá.October1992.

[CAM 96] Campo, M.; Price, R. *A Reflective Framework for Software Visualization Tools*. Procs. of the 10th Brazilian Symposium on Software Engineering, São Carlos, Brazil. October 1996. (In Portuguese)

[CAM 97] Campo, M. *Visual Comprehension of Frameworks through Introspection of Examples*. Ph.D. Thesis. UFRGS, CPGCC, 1997. (In Portuguese)

[CHI 90] Chikofsky, E.; Cross, J. *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Software, v.7, n.1, p. 13-17, Jan. 1990.

[DEP 93] De Pauw, W.; et al. *visualizing the Behavior of Object-Oriented Programs.* ACM Sigplan Notices, v.28, n.10, New York:ACM Press, p.326-337, Oct.1993.

[DEP 94] De Pauw, W.; et al. Procs. ECOOP'94, 10, 1994, Bologna, Italy. Berlin:Springer-Verlag, 1994. p. 175-194.

[DEU 89] Deutsch, P. *Frameworks and reuse in the Smalltalk-80 system*, In: Biggerstaf, T., Perlis, A. (Eds.) Software Reusability: Applications and Experience, New York: ACM Press, 1989.

[FIS 91] Fisher, G.; Henninger, S.; Redmiles, D. *Cognitive Tools for Locating and Comprehending Software Objects for Reuse.* Procs. Intl. Conf. on Software Engineering, 13., 1991. Los Alamitos: IEEE Press, 1991. p. 318-328.

[GAM 93] Gamma, E.; et al. *Design Patterns: Abstraction and Reuse of Object-Oriented Design.* Procs. ECOOP'93, Kaiserslautern, Germany. Berlin:Springer-Verlag, 1993.

[GAM 94] Gamma, E.; et al. *Design Patterns: Reusable Elements of Object-Oriented Design*, Reading: Addison-Wesley, 1994.

[GOL 83] Goldberg, A. *Smalltalk-80: The Language and its Implementation.* Reading:Addison-Wesley, 1983.

[HEL 90] Helm, R.; et. al. *Contracts: Specifying Behavioral Compositions in Object Oriented Systems.* Procs. OOPSLA'90, Otawa, Canada. New York: ACM Press, 1990.

[JER 96] Jerding, D.; Stasko, J.; Ball, T. *Visualizing Message Patterns in Object-Oriented Program Executions.* Georgia Technology Institute, 1996. (Tech. Report GIT-GVU-96-15).

[JOH 88] Johnson, R.; Foote, B. *Designing Reusable Classes.* Journal of Object-Oriented Programming, v.1, n.12, 1988.

[JOH 92] Johnson, R. *Documenting Frameworks Using Patterns.* Procs. OOPSLA'92, Vancouver, Canadá. New York:ACM Press, Oct.1992.

[JOH 93] Johnson, R. *How to Design Frameworks*, OOPSLA'93, Washington DC. Tutorial Notes.1993.

[KRU 92] Krueger , C. *Software Reuse.* ACM Computing Surveys, v.24, n.2, June 1992.

[LAJ 94] Lajoie, R.; Keller, R. *Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert.* Procs. 62 Congress of the Association Canadienne-Française pour l' Avancement des Sciences, Montreal, Canada, 1994.

[LAN 95] Lange, D.; Nakamura Y. *Interactive Visualization of Design Patterns Can Help in Framework Understanding.* ACM Sigplan Notices, v.30, n.10. Oct. 1995.

[LOR 94] Lorenz M, Kid, J. *Object-Oriented Software Metrics - A practical guide.* Englenwood Cliffs : Prentice Hall. 1984

[MAE 88] Maes, P. *Issues in Computational Reflection.* In: Meta-Level Architecture and Reflection. Amsterdam: Elsevier Science, 1988.

[MAY 95] Mayrhauser Von, A.; Vans, A. *Program Comprehension During Software Maintenance and Evolution.* IEEE Computer, v.24, n.8, p. 44-55. Aug. 1995.

[OPD 92] Opdyke, W. *Refactoring Object Oriented Frameworks*. Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.

[PAR 79] Parnas, D. *Designing software for ease extension and contraction*, IEEE Transactions on Software Engineering, v.5, n.2, p. 128-137, Feb 1979.

[PRE 94] Pree, W. *Meta-Patterns: Abstracting the Essentials of Object-Oriented Frameworks*, Proc. ECOOP'94, Bologna, Italy. Berlin:Springer-Verlag, 1994.

[RUM 91] Rumbaugh, J.; et al. *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[SOL 84] Solloway, E; Ehrlich, K. *Empirical Studies of Programming Knowledge*. IEEE Transactions on Software Engineering, v.10, n.5, p. 595-609, May. 1984.

[STA 90] Stasko, J. *Simplifying Algorithm Animation with TANGO*. Procs. IEEE Workshop on Visual Languages, 1990, Stockie, Illinois, 1990.

[STA 94] Stasko, J.; Jerding, D. *Using Visualization to Foster Object-Oriented Program Understanding*. Atlanta, Georgia Institute of Technology, July 1994. (Technical Report GIT-GVU-94-33).

[TAE 89] Taenzer, D.; Ganti, M.; Podar, S. *Object-Oriented Software Reuse: The Yo-Yo Problem*. Journal of Object-Oriented Programming, v.2, Set. 1989.

[TIL 96] Tiley, S; Santana, P. *Towards a Framework for Program Understanding*. 4th IEEE International Conference on Program Comprehension, Berlin, Germany, 1996.

[VIO 94] Vion-Dury, J.; Santana, M. Virtual Images: Interactive Visualization of Distributed Object-Oriented Systems. Procs. OOPSLA'94, Portland, Oregon, 1994.

[Wil 92] Wilde, N.; Huit, R. Maintenance Support for Object-Oriented Programs, IEEE Transactions on Software Engineering, v.18, n.12, Dec.1992.