A Reverse Engineering Approach to Framework Comprehension

Marcelo Campo

UNICEN-Fac. Cs. Exactas-ISISTAN
San Martín 57
Tandil, Bs. As., Argentina
e-mail: mcampo@necsus.com.ar

Tom Price

UFRGS- Instituto de Informática
Caixa Postal 15064
Porto Alegre - RS- Brasil
e-mail: tomprice@inf.ufrgs.br

Abstract

Framework comprehension is a very limiting factor to take full advantage of the benefits that frameworks offer to increase quality and productivity in software development. In this paper, a reverse engineering approach to framework comprehension using the MetaExplorer tool is presented. MetaExplorer is characterized by the use of meta-objects to gather information from framework applications, providing a rich set of visualizations, and abstraction capabilities for subsystem analysis and design-patterns recognition, along with advanced exploration mechanisms based on semantic zooming and direct-manipulation user interfaces. The effectiveness the tool to help on the process of framework understanding was tested through controlled experiments, whose metrics suggest that users of the tool grasp a much better understanding of an analyzed framework than users not using the tool.

Keywords: frameworks, design patterns, software comprehension, software visualization, meta-object models

1. Introduction

Program comprehension is one of the most critical problems in the software life cycle. The success of activities such as debugging and maintenance depends, in a great level, on how easy it is for a programmer to understand a given program when it is necessary to correct an error or change its functionality. This problem also affects the reusability of a software artifact. The reuse of software artifacts is characterized by the need of changes or adaptations on the artifact being reused, in order to make it adequate to the new application requirements [KRU 92]. Deutsch suggests that software reuse exists only when exists some change, either in the reused artifact or the context where it is used [DEU 89]. In this way, independently of the reuse technology (source code components, software skeleton, very-high-level languages, etc.), a programmer must *understand* the nature of the abstractions in order to select among the available components those that fulfill the new application requirements, as well as to make the needed modifications or extensions with specific functionality.

Object-oriented programs are, in general, more difficult to understand than traditional procedural systems (with functional architecture). The well-known problems of the dichotomy

between the source code structure and the execution model, the distribution of functionality among classes and the dynamic binding, makes object-oriented programs harder to understand [WIL 92][DEP 93][LAN 95].

This complexity is even more acute in the case of frameworks [JOH 88][DEU 89]. Essentially, a framework is constituted by a set of abstract classes that implements a domain specific architecture [BEC 94]. Framework abstract classes provides 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 was 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 a framework is designed by domain experts, users of the framework are reusing, implicitly, the experience of these experts.

Due to these characteristics, frameworks offers 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. In order to be able to adequately specialize abstract classes and to describe the best way an application can be built through the composition of instances of subclasses of those classes, a user is often faced with the need of comprehending the detailed design of 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 show performance problems or they are limited to very restricted domains [PAR 79]. Flexible solutions can be adapted to particular cases, allowing programmers to exploit those aspects that simplify the solution, in terms of performance and functionality. Complex frameworks often 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. In general, flexible design structures implies very complex designs and, in consequence, designs harder to understand.

In this context, even if documentation is available, visual tools to help a framework user to analyze the structure and behaviour of applications built using a framework at different levels of abstraction become a valuable complement to facilitate the process of framework comprehension. Particularly, reverse engineering tools able to recognize and visualize abstractions of higher level than the source code, such as subsystems and design patterns [GAM 94], provide an excellent vehicle for understanding a given framework in a higher level of abstraction than simple visualizations based on classes or interactions among objects.

In this paper, a reverse engineering approach to the problem of framework comprehension is presented. The approach is based on the Luthier framework [CAM 96][CAM 97] for building tools for application analysis and visualization through reflection techniques based on meta-objects. Luthier provides flexible support for building visualizations tools adaptable to different analysis functionality, using a hyperdocument manager to organize the collected information. These mechanisms supports the flexible construction of different visualizations from the analyzed examples, as well as the navigation among different visual representations and textual

Comentario [DRM1]: Um conjunto de classes que fornecem uma solução genérica para um dado domínio de aplicação é denominado de ***framework orientado a objetos*** [DEU 89][JOH 88].

Comentario [DRM2]: Um framework é constituído por um conjunto de classes que definem um projeto abstrato para soluções de problemas de uma família de aplicações dentro de um domínio [JOH 88]. Um *framework* implementa, em termos de classes, o comportamento genérico de um domínio de aplicação, deixando a implementação dos aspectos específicos de cada aplicação para serem completados por subclasses.

Comentario [DRM3]: O problema habitual que os usuários de um framework encontram é que, para conseguir especializar as classes abstratas e descrever como a aplicação se constrói a partir dessas classes, precisam geralmente compreender o projeto detalhado das classes do framework. Em frameworks complexos, estas classes codificam padrões de colaboração entre instâncias, através de estruturas de projeto *flexíveis*, isto é, estruturas que permitem *adaptar* o comportamento *geral* provido pelo framework.

documentation, through the explicit support for editing documentation books. With this support a prototype of a visual tool for framework comprehension, MetaExplorer, was developed. MetaExplorer provides a rich set of visualizations, and abstraction capabilities for subsystem analysis and Gamma design-patterns recognition. The effectiveness of the approach to help on the process of framework understanding was tested through controlled experiments, which suggest that users of the tool grasp a much better understanding of an analyzed framework than users not using the tool.

The paper is organized as follows. The next section discusses the different aspects that contribute to make difficult the process of framework comprehension and the role that architectural abstractions play in the design of support tools. In section 3 a brief description of the developed comprehension tool and the results of controlled experiments are presented and analyzed. Section 4 presents the most relevant mechanisms provided by the Luthier framework. Finally, section 5 discusses related works in the area, and section 6 outlines the main conclusions extracted from the project.

2. Framework Comprehension and Architectural Abstractions

Software tools for program comprehension are of great importance to help to reduce the inherent complexity of the comprehension process. These tools aim to help the programmer to build a mental model of the program, by providing mechanisms for analyzing, exploring and visualizing information about the program at different abstraction levels. Frequently they provide different visual representations that synthesize relevant properties about the analyzed artifact, as well as mechanisms for filtering, organizing and abstracting information, which allow the user to explore the information from different perspectives. Reverse engineering and software visualization systems are the lines of research that have made the most important contributions to the development of software comprehension support techniques. Reverse engineering is the activity through which relevant information about a program or system is identified, relationships are discovered and abstractions are generated [CHI 90]. Tools to support this activity aim to provide automatic mechanisms to extract the information from a program by analyzing its source code, and to deduce or recognize structural and behavioral abstractions not directly represented in that code. Reverse engineering tools have shown its usefulness to aid the process of software comprehension, particularly in the case of legacy systems, but they also offers an interesting alternative to help in the reuse of object-oriented systems and particularly frameworks.

A framework is, essentially, the implementation, in terms of classes, of a generic architecture for an application domain [BEC 94]. A previous knowledge about the application domain is, doubtless, of great importance to help in a given framework comprehension. Through a general knowledge of the domain, a programmer is able to comprehend the general organization of concepts or, more specifically, the domain model implemented by the framework. On the other side, 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 need to have a deep knowledge about the application domain, but just the needed 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 only the functionality that abstract classes leave to be implemented by subclasses.

Comentario [RJO4]: Um framework é, essencialmente, a implementação, em termos de classes de uma arquitetura genérica para um domínio de aplicação. Um Um conhecimento prévio do domínio de aplicação é, sem dúvida, de grande importância para facilitar a compreensão de um dado framework. Através do conhecimento geral do domínio, um programador pode compreender a organização geral de conceitos, ou, mais especificamente, o modelo de domínio que o framework implementa. Por outro lado, também é necessário ter em consideração que o objetivo do desenvolvimento de um framework é permitir a reutilização, por parte dos usuários do framework, do conhecimento de domínio que o projetista possui. Assim, é razoável esperar que os usuários do framework não necessitem possuir um conhecimento profundo do domínio de aplicação, mas só o conhecimento necessário da funcionalidade da aplicação que deseja-se implementar. Idealmente, para ser realmente de utilidade, um framework deve permitir ao usuário a implementação de aplicações partindo do conhecimento da funcionalidade que as classes abstratas deixam para ser implementada por subclasses.

In this context, a reasonable first step on a framework comprehension process is to provide the user the mechanisms to allow her to build an initial mental model of the structure and behaviour 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 make easier the global comprehension of functionality of a framework.

Subsystems and design patterns [GAM 94] represent design abstractions that are not supported by current object-oriented languages, but are of great importance to comprehend 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. If a user knows which is the problem that a given pattern is intended to solve, and how the classes and methods that the pattern prescribes for the generic solution, then such 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 make easier the comprehension of how determined parts of the framework were designed and the function that some methods play inside a given class.

It is necessary to take into account that an approach centered exclusively 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.

In spite of this, a reverse engineering approach based on the recognition of these functional units in a framework becomes an important complement to provide the user with more abstract initial views of the framework organization. Through adequate mechanisms that enable the framework visualization and, essentially, exploration at different levels of abstraction a user can get an overall comprehension of the framework that can be gradually refined though a further analysis of the behaviour at the instance level, whenever this analysis is necessary.

3. Experimenting Reverse-Engineering-Based Techniques

In the past three years, the author have been involved in a research project on the development of a framework, called Luthier, to build visual tools to help the process of framework understanding. As a result of the work, a Smalltalk prototype, called MetaExplorer, was developed, which provides a rich set of features that enables the analysis and visualization of a framework at different levels of abstraction from both architectural and instance behaviour points of view. In this section the main capabilities of MetaExplorer are briefly described and the results of its experimental application are presented. The next section presents a more detailed discussion of the mechanisms provided by the Luthier framework for the construction of such tool.

Comentario [RJO5]: Assim, um primeiro passo razoável no processo de compreender um framework é prover o usuário com mecanismos que lhe permitam construir um modelo mental inicial da estrutura e comportamento da arquitetura implementada por esse framework. Neste sentido, a provisão de suporte para o reconhecimento abstrações que não são explicitamente suportadas pela linguagem de programação, é um complemento importante para facilitar a compreensão global da funcionalidade de um framework.

Comentario [RJO6]: Os padrões de projeto [GAM 94] representam abstrações de projeto que não são suportadas pelas linguagens orientadas a objetos atuais, mas que são de grande importância para compreender como os objetos de um sistema são organizados e colaboram para satisfazer a funcionalidade global.

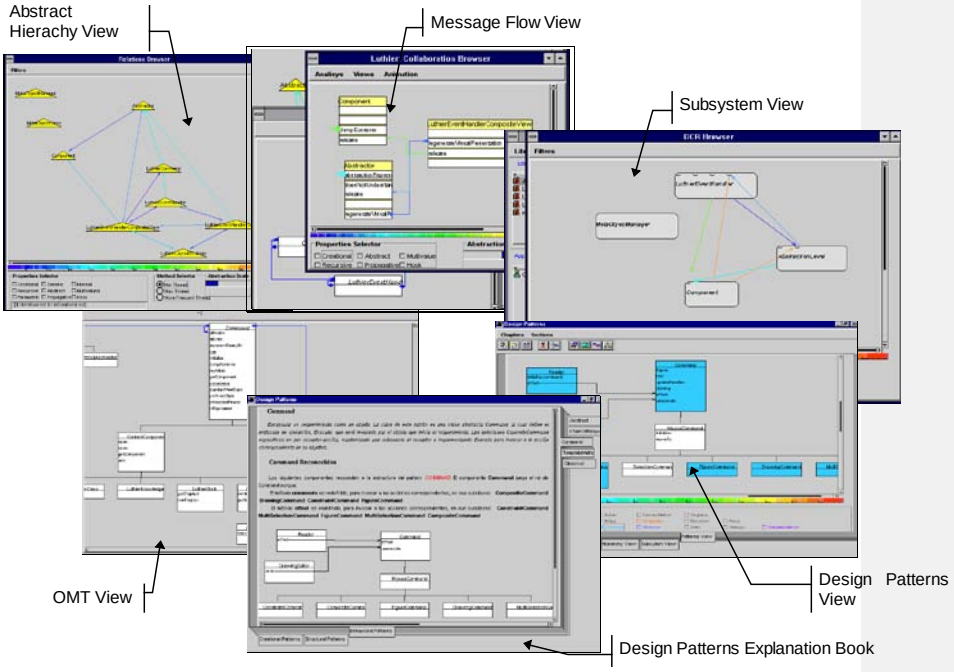

Fig. 1- Different visualizations provided by the tool

3.1 MetaExplorer Overview

MetaExplorer is characterized by the use of meta-object-based reflective techniques to analyze the static structure and dynamic behaviour of applications built using a given framework. Meta-objects [MAE 88] provide an excellent vehicle to gather both static and dynamic information, allowing to build dynamically-configurable tools for program analysis in an uniform object-oriented model.

The first step to analyze a given framework with MetaExplorer is the reflection of the classes to be inspected on a pre-defined set of meta-objects. Next the application or example is executed and, through a specially designed meta-architecture, an abstract representation of the framework is generated using an advanced hyperdocument manager. This manager implements a model of aggregate nodes that can persistently store the collected information on the behaviour of the observed framework in abstracted collections defined by the tool implementor. This representation provides the support for navigating among multiple visual representations and documentation books constructed by the user during the comprehension process as well as generated automatically by the tool.

From such representation different potential abstractions, such as subsystems and Gamma design-patterns, are automatically recovered, and different visualizations of the framework static and dynamic structure are produced using conventional notations, as OMT, message flow graphs, object interaction charts, etc. (Fig. 1). Also, MetaExplorer provide alternative abstract

visualizations as abstract hierarchy graphs and message-flow graphs highlighting characteristic aspects of frameworks as for example method categories (i.e., abstracts, templates, hooks and base).

From any of these visualizations, specific meta-objects can be dynamically associated to application objects to enable the detailed analysis at instance level (i.e., visualization of internal state changes, breakpoints on methods, and so on). This functionality allows the user to focus on specific points that require a detailed analysis of the involved instances, avoiding the collection of full information about instances. This is one of the most problematic points in object-oriented program visualizations tools due to the huge amount of information that can be gathered in a single execution. In this way, MetaExplorer induces a two-phase exploration process oriented by the visualization of high-level architectural representations.

Navigation and direct-manipulation user interfaces are two complementary mechanisms to facilitate the exploration of complex data. MetaExplorer provides a powerful direct-manipulation user-interface, which allows, for example, a rubberband selection and zooming of specific portions of diagrams using alternative notations, as well as the animation of the framework message flow (in message-flow-based visualizations) under interactive control of the user [CAM 96]. MetaExplorer also implements an innovative mechanism for semantic zoom [MUT 95] based on symbolic abstraction scales. This mechanism allows the user for the continuous zooming of diagrams, showing or hiding pertinent information at each level of abstraction (this feature is described in the next section). This zooming greatly helps to reduce the proliferation of multiple windows, contributing to avoid the well-known problem of user disorientation in hypertext systems.

Interactive filtering of information through query capabilities is provided as an additional mechanism to reduce the complexity of visualizations. Textual queries, based on abstract message flow properties, allow the user to filter the visualization to show only those components that satisfies some abstract properties, as for example, classes related by messages that activate redefinition of abstract methods. Presentations are enhanced by the use of colors to suggest a relative sequence in which relationships are established at runtime. The semantic of this colors can be interactively selected by to user, to represent the first, the more frequent or the last message that determined a relationship between two classes. This feature enables the analysis of dynamic relationships at architectural level.

One important aspect, missing in most of the visual understanding tools for object-oriented programs currently reported in the literature, is a support to explain the structure and behaviour of the analyzed program. Also, these tools provides little or no support to produce additional documentation product of the comprehension process carried out by a user. For this task MetaExplorer provides support for the construction of documentation books fully integrated with the visualizations produced from the captured program information. Books can be organized in terms of chapters and sections, allowing the interactive use of visual attributes and fonts to highlight title paragraphs 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.* Each chapter is composed of as many sections as patterns in each category were recognized. A section includes a short explanation about the pattern that it describes, and textualy explains the reasons that suggested the existence of one or several occurrences of the pattern in the framework structure. The

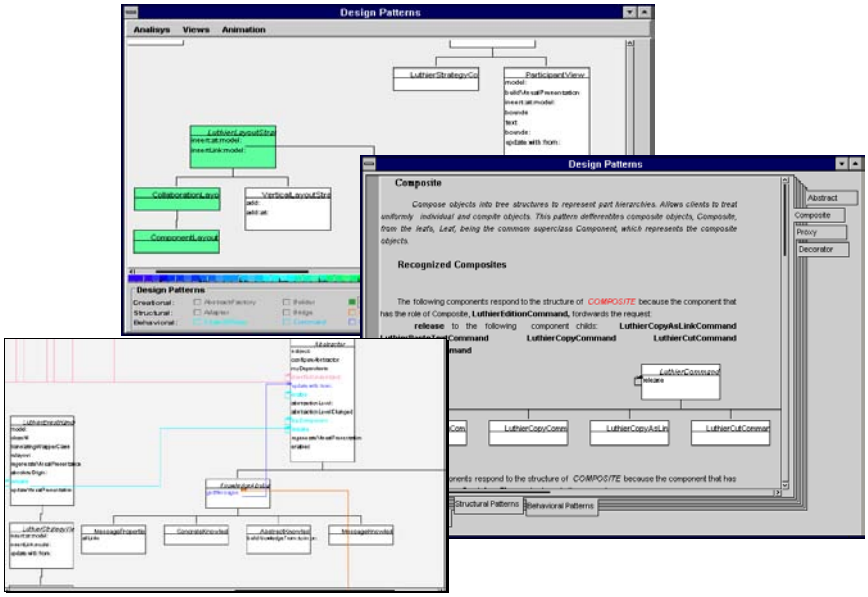

Fig. 2- Alternative visualizations of design patterns and documentation book

explanation includes an 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 design pattern. Through the navigation capabilities the user can navigate among the different visual representations up to the implementation of the methods. This functionality allows the framework exploration at different levels of abstraction, starting at the abstraction levels provided by subsystems and design patterns.

Fig. 2 shows snapshots of the graphical browsers provided by MetaExplorer to visualize design patterns recognized in the structure of an analyzed framework and the page for the Composite design pattern of the explanation book automatically generated by the tool. 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.

3.2 Experimental Results

The effectiveness of a support tool for framework comprehension can only be empirically demonstrated through its use in real scenarios. For this reason, controlled experiments with three groups of students were made to evaluate the effectiveness of the different techniques provided by MetaExplorer to aid the process of framework understanding.

One experiment consisted of the implementation of a graphical editor for PetriNets using the HotDraw framework [JOH 92]. The experiment was designed to allow the extraction of conclusions about the overall usefulness of the tool, as well as the specific advantages that reverse engineering techniques for abstraction recovery could provide a user to facilitate the comprehension process. One group, called GS, was provided with a version of the tool that only supported structural visualizations with animation of message flow capabilities. A second group, GA, was provided with the same support plus visualizations of subsystems and design patterns, while a third group, called GN, did not use any tool to understand the HotDraw framework.

The editor all groups had to develop was specified to satisfy exclusively the following functionality:

- Figure creation: *Transitions*, with rectangular shape and two bounded texts, the name (inside the rectangle) and the transition condition (external to the rectangle). *States*, with elliptical shape and two internal texts. *Connections*, represented by an arrow relating one state with one transition and vice-versa, with an associated text representing the parameters of the transition.
- Direct manipulation for creating, moving and resizing figures and basic editing commands with undo (i.e., insertion, deletion, etc.).
- Visual Feedback for the creation of connections indicating the valid targets of a new connection starting at a given transition or state.

The definition of figures to be edited and the use of the constraint system were the most important design aspects of the required editors, because most of the needed functionality is provided by the framework. In this way, it was possible to analyze the degree of framework comprehension in the aspects of redefinition of abstract classes and the reuse of finished components.

The developed applications were compared using a tool for metrics collection also built using the Luthier framework. This tool implements the metrics described in [LOR 94], from which, only a relevant subset to analyze differences among applications developed using a framework was considered for evaluating the results. Table 1 presents the values corresponding to the three groups. These values admit several interpretations, as they strongly depend on the characteristics of the analyzed program. Nevertheless, they offer interesting suggestions about relative differences among the three developed applications, which can be further checked through a deeper analysis using the understanding tool. Values highlighted in the table represent values that are particularly interesting as a suggestion of the reuse level of the framework.

The values shown in the table make evident a substantial difference in the number of classes created by the groups using the tool and the group not using it. This represents a strong suggestion of a poorer comprehension of the framework by the GN group. Also, not too many differences can

Metric	GS	GA	GN
Totals			
Number of Classes	**16**	**12**	**32**
Lines of Code	1427	980	1854
Hierarchy Nesting Level (max.)	7	7	8
Number of Methods	**178**	**148**	**281**
Number of Overridden Methods	102	95	126
Number of Added Methods	**76**	**53**	**155**
Number of Sentences	700	718	1217
Number of Messages	**815**	**781**	**1370**
Number of Class Variables	**0**	**0**	**16**
Number of Instance Variables	23	12	15
Number of not Called Methods	**51**	**25**	**115**
Number of not Called Public Methods	94	**75**	125
Number of not Called Private Methods	33	38	41
Averages			
Number of Inherited Methods/Class	127.50	**130.75**	127.56
Number of Overridden Methods/Class	**6.37**	**7.91**	3.93
Number of Added Methods/Class	8.56	4.41	6.85
Proportion Overridden/Added	0.63	**1.55**	0.61
Specialization Index	**3.47**	**3.20**	**2.36**
Lines of Code/Method	8.215	**6.62**	7.23
Number of Messages/Method	**4.58**	5.27	4.87

Table 1- Values of Some Relevant Metrics

be appreciated between the GS and GA groups, particularly in the amount of lines of code and the number of added methods. The specialization index (Hierarchy Nesting Level * Number of Methods / Number of Added Methods), provides a clue about the level of reuse of a class hierarchy relative to the number of added methods. The values for this index for groups GS and GA 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, GS presents the greater number of new methods per class and the higher nesting level. In this case, the specialization index is complemented with a great number of inherited methods.

It is necessary to take into account that, in general, it is considered that an adequate design should add behaviour in subclasses, redefining a few methods. This is not necessarily true with well-designed frameworks, in which few additional methods are supposed to be necessary. Obviously, this consideration is relative to each particular framework, and it also depends on the inheritance or compositional design style that predominates in the framework design. However, in order to compare applications, a relation between the number of overridden and added methods can be useful as a suggestion of the reuse level obtained by different applications, especially, if the framework is an inheritance-based one. In this case, this relation should tend to infinite for an *ideal* framework, in which any application could be produced by implementing just abstract and hook methods (a class could add private methods for internal design decisions, which do not extend the protocol of the framework). For this reason, the *Proportion of Overridden/Added* metric was included to represent this value. A higher value of this relation can suggest a greater reuse of the

framework. The values shown in the table for this metric show a high parity between groups GS and GN, while the value for GA is almost twice in magnitude. The combination of both indexes suggest a better degree of reuse for group GA.

These values suggest a better performance of group GA, but they do not indicate a solution of higher quality nor reuse, between groups GS and GA. In order to determine which was the better solution, a detailed analysis about design differences that these metrics do not reflect, was necessary. Analyzing the applications using MetaExplorer, similar information to that provided by the metrics was extracted: the application developed by group GN presents greater problems related to the design of the editor, while the other two applications show little difference between them. Comparing the results in a general way, the main differences among the design decisions of the three groups are related to the design of the figures to be edited and the utilization of the constraint system:

- GS presents a better utilization of the constraint system, which allowed them to easily solve some problems that arose due to bad decisions about class specialization.
- GA 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 the behaviour implemented by the framework, mainly on the aspects related with abstraction design as well as on the use of the constraint system.

3.2.1 Development Times

In order to evaluate the impact of the tool usage, registrations of the time spent on developing the editors were taken. Table 2 presents times used by each group, according to the time taken up exclusively by comprehension activities and the time involved in application development.

These times present interesting data that complete and explain some of the differences shown above. As can be seen, group GS dedicated more time to comprehension activities and less time to development, while GA was the group that used less total time. The difference between the times of tool use is the reason for the much better use of the constraint system by group GS. This group preferred to postpone the development phase until achieving the total comprehension of the framework; GA, on the other hand, having more abstract visualizations that guide the exploration of the framework used the tool to achieve a global comprehension and to analyze partial aspects that could not be solved during the programming phase. A similar strategy was also used by GN, which analyzed the examples through code inspection, and had similar comprehension times as GA.

Group	Comprehension Activities (Hs)	Development (Hs)	Total Time (Hs)	
GS	72	100	172	
GA	25	108	133	
GN	26	131	157	
Average/ Deviation	41 / 26.8	113 / 16.1	154 / 19.6	

Table 2- Time Used in Application Development

3.2.2 Conclusions from the Experiment

Some relevant conclusions can be outlined from the experiment described above about the utilization of MetaExplorer.

In a first place, the experiment suggest that the use of MetaExplorer helped to obtain better results, in terms of the reuse of the functionality provided by the used framework and quality of the developed editors. This result, combined with the similar development times, suggest an important gain in terms quality of the final solutions. Considering the differences between both groups using the tool, the total time used by GA group suggest that the exploration based on abstractions as subsystems and design patterns induced more adequate design decisions, making the framework exploration easier and more productive.

On the other side, it was noticed that the tool can induce an exaggerated exploration of details that are not necessarily relevant. Nevertheless, the comprehension of these details helped the group GS to use a very complex subsystem, such as the constraint system, very well.

Certainly, these conclusions cannot be considered as definitive because of the small size of the sample and the narrow scope of the experiment. However, they empirically confirm the hypotheses that an abstraction-oriented exploration strategy, complemented by navigation capabilities among alternative representations can effectively help inexperienced users to better understand a framework, and therefore to make a better reuse of such framework.

4. An Overview of the Luthier Framework

MetaExplorer was developed using the Luthier framework [CAM 96] designed, and implemented in VisualWorks-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 Maes-style meta-objects [MAE 88].

Luthier is constituted by four sub-frameworks, which provides adaptable support for the four essential tasks that characterize both reverse engineering tools and visualization systems, that is, LuthierMOPs for information gathering, LuthierBooks for information representation, LuthierViews for visualization and exploration of gathered information and LuthierAbstractors for abstraction analysis and recovery.

Fig. 3 presents the generic structure of a visualization tool built using Luthier. A typical tool built using Luthier will be composed of a set of meta-objects monitoring the execution of an application, generating an abstract representation of the program information using a specific hyperdocument manager. The visualizations will request at the lower level the information to be visualized, which will be provided by *abstractor objects*. Abstractors are in charge of recognizing

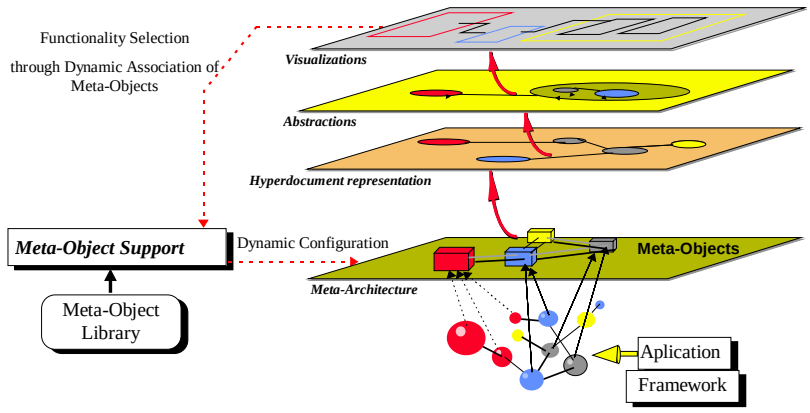

Fig. 3- Generic structure of a visualization tool built using Luthier

or building abstractions from the information contained in the hyperdocument representation. The next sub-sections briefly describe the four sub-frameworks emphasizing the most relevant contributions of Luthier: *meta-object managers* and *abstractors*.

4.1 LuthierMOPs: Customizable Meta-Object Protocols

A distinctive characteristic of Luthier is the sub-framework for meta-object support based on the concept of *meta-object managers* [CAM 96]. A meta-object manager (MOM) is an object which determines how meta-objects are associated to base-level objects and how these meta-objects are activated. 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.

This approach presents two main advantages from a program analysis tools point of view:

- ***Activation strategies***: Operationally, a MOM acts as a *mediator* between base-level objects and meta-objects. Messages reflected from the base-level are directed towards a given MOM which decides what meta-objects, if any, must be activated. In this way, MOMs can support different strategies for meta-object activation, as for example, priorities of activation when several meta-objects are associated with the same object.
- ***Association policies***: MOMs provides a greater level of flexibility to encapsulate in different objects different *policies* of meta-object association. This allows the separation of specific aspects of meta-object functionality from the aspects related to their organization in a meta-architecture. For example, a MOM can implement the association of a single meta-object to a given class, in such a way that meta-object be activated if a given instance does not has its own meta-object. Alternatively, another MOM may allow

the association of multiple meta-objects to a given object, or even to restrict the association of one meta-object per object.

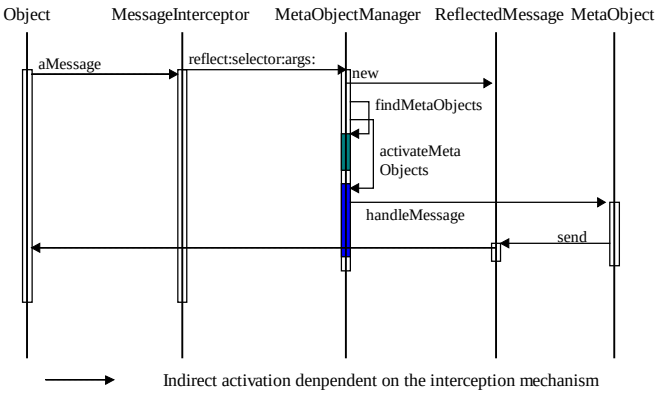

Fig. 4. Abstract controlflow of *LuthierMOPs*

From a point of view of tool construction, MOMs offer the advantage of providing a high-level interface to organize meta-objects independently of the functionality implemented by them. The separation of the association mechanism allows for its specialization to implement specific management services adequated to the requirements of each particular tool. This ability is essential to enable the interactive substitution of meta-objects which allows the comprehension tool to switch among different data gathering functionality. The ability to activate meta-objects, enables a MOM to dynamically suspend and restart the reflection of messages of all the reflected objects, or just some of them. This capability allows, for example, the user to interactively determine whether a tool should activate meta-objects. Also, different functions of the meta-level can be activated or deactivated without accessing any specific meta-object.

LuthierMOPs defines four abstract classes which reify the different aspects involved in the implementation of a meta-object support. The abstract interaction among these classes is presented in Fig. 4. When a reflected object receives a message, this message is deviated by the interception mechanism to the associated MOM. The MOM looks for meta-objects associated to the object that reflected the message (*findMetaObjectsFor*: message) and decides whether to activate the selected meta-objects (*activateMetaObjects:* message) by sending to them the *handleMessage: manager*:. message. When a meta-object is activated receives an instance of the *ReflectedMessage* class which contains all the information relative to the reflected message. The meta-object can execute the original method by sending to the ReflectedMessage instance the *send* message.

Customized meta-object management mechanisms can be easily implemented by providing specific implementations for the *findMetaObjectsFor* and *activateMetaObjects* messages. Different meta-objects must provide the implementation of the *handleMessage:manager*: method.

4.2 LuthierBooks: Information Representation

The LuthierBooks sub-framework provides support to define specific representation of the gathered information in terms of an hypertext model based on contexts objects. Contexts define

aggregate nodes whose semantic is defined by the class that implements it. The model supports classes of nodes and bi-directional links that make easier the navigation through the complex web of information determined by the execution of an object-oriented program. This representation can be stored persistently as part of documentation books. These books, are implemented in terms of the same hypertext model, enabling the constructions of design libraries organized as an hyperdocument.

4.3 LuthierViews: Information Exploration

The LuthierViews sub-framework provides the common infra-structure to implement dynamically configurable visualizations with direct-manipulation user interfaces. This sub-framework is an extension of the MVC framework, providing facilities to build direct-manipulation zooming mechanisms using alternative visualizations for the selected information. *LuthierViews* also provides components to create books with formatted text through user-defined styles, text editing capabilities and insertions of visual components generated by visualizations, and even complete visualization tools, as part of a standard page of a book. This functionality enables a better organization and enhanced visual presentations of the framework information.

4.4 LuthierAbstractors: Materializing and Managing Software Abstractions

Luthier introduces the concept of *abstractor objects*, which explicitly separate the information representation from visualizations (Fig. 5). Abstractors represents a generic architectural component of tools, by which different analytical algorithms and selection criteria can be located, without the need of modificating either classes of the information representation or classes implementing visualizations. Following the conventional communication mechanism among views and models in MVC, views request their model the information to be visualized through direct messages. An abstractor substitutes a model and decides whether to ask the original model for the requested information. In this way an abstractor can be designed to behave in three different ways:

- Information generation: an abstractor can encapsulate the algorithms to recognize subsystems, collaborative groups or design patterns and to provide such abstractions as normal data to be visualized.
- Information selection: an abstractor can be designed to select information from the model, according to some criteria established either internally or externally. Selectors allows for the specification of selection criteria, as for example, to visualize only classes that are related through messages that activate abstract methods in redefined in subclasses. These selections can be externally defined by the user, providing in this way the ability to vary the detail level of a given visualization according to the type of information a user wants to focus, at any time of the exploration.
- Information filtering: an abstractor can decide whether a given data element will be visible or not, making it available or not to the visualization. In this way, through the use of abstractors, visualizations must only deal with the graphical presentation of the information to be shown, without take into account the necessary detail level. This greatly

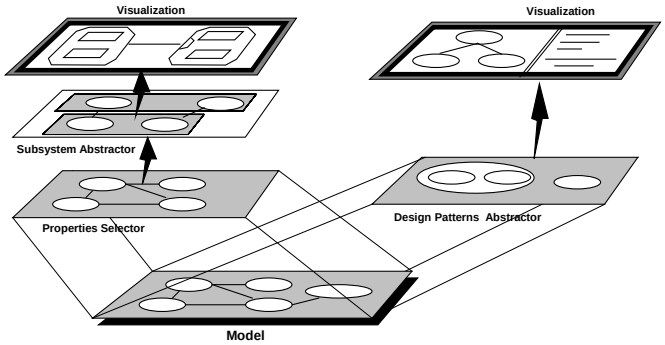

Fig. 5- Relationship among visualizations, abstractors and information representation

reduces the complexity of programming new visualizations, which is, perhaps, the most time-consuming task in the construction of visualizations tools.

Essentially, abstractors behave as *proxies* of the objects contained in the model, controlling the access to these objects by visualization classes. In this way, abstractors can be hierarchically composed to provide independent control over each object contained in the model. Also, they can be dynamically composed to combine different functionality, as for example, filtering on a specific selection of the program information. This powerful feature enables the combination and reuse of different algorithms for abstraction recognition with different visualizations styles, as for example, subsystem analysis, structural relationship analysis and design patterns.

4.4.1 Abstraction Scales

LuthierAbstractors provide the generic support for managing symbolic abstraction scales, which enable the semantic zoom of visualizations without the need of programming special filters in visualizations. An abstraction scale is an ordered tuple naming the order in which constructions, like subsystems, classes, methods, and so on, should be visualized. An 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. showing or hiding dynamically details). The visualizations, in turn, only have to worry about what must be shown according to the data that abstractors pass to them, in the current abstraction level. For example, the scale below is used to define the different detail levels in which a subsystem-based visualizations can be shown.:

(subsystemAbstracion abstractHierarchy abstractMethod concreteMethod concreteHierarchy)

A selection of a level in this scale will define which information the visualization will receive to be graphically presented. That is, if the selected abstraction level is *abstractHierarchy* the visualization will only receive the subsystems and the top of each component class hierarchies. After that if the user selects *abstractMethod*, the same visualization will receive subsystems, abstract classes and the abstract methods defined in such classes (Fig. 6).

The scale below defines a scale in which methods are shown only in the last (or higher) level of detail:

(subsystemAbstracion abstractHierarchy concreteHierarchy abstractMethod concreteMethod)

LuthierAbstractors provides the generic mechanisms that implement this behaviour, and the standard protocol through which visualizations request information to be visualized. Each abstractor object has its own instance of abstraction scale, so it is possible to vary independently the abstraction level of each abstractor representing data to be visualized.

Each view asks its model information to be visualized though two standard messages *getNodes* e *getLinks*. The generic behaviour of this messages implements the control mechanism of the current abstraction level. If such level is greater than the level represented by the abstractor, the complete information of the model is returned. Otherwise only the corresponding abstract information is returned:

```
getNodes
        self abstractioLevel > self abstractionRepresented
                ifTrue:[ ^self getFullNodesInformation]
                ifFalse:[ ^self getAbstractNodesInformation].

getLinks
        self abstractioLevel > self abstractionRepresented
                ifTrue:[ ^self getFullLinkInformation]
                ifFalse:[ ^self getAbstractLinkInformation].
```

The default implementation of *getFullNodesInformation* method return the full component list of the abstractor, while *getAbstractNodesInformation* returns an empty list indicating that there is no information to be visualized at the current level of abstraction. This mechanism can be specialized in subclasses to implement, for example, semantic zoom mechanisms based on the current attention focus of the user. In the case of hierarchically-composed abstractors, the abstraction level is common to all the component abstractors, and changes produced in the upper levels are automatically propagated to the lower ones.

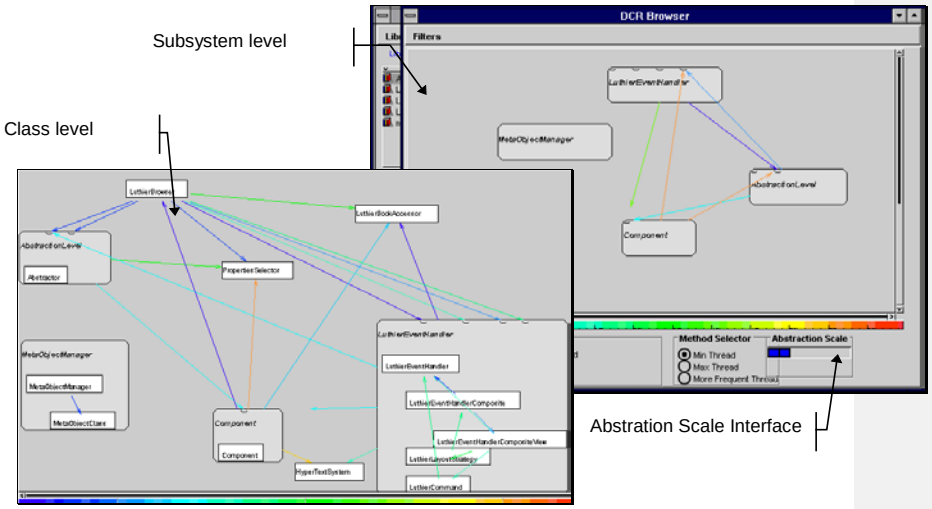

Fig. 6- Example of an interactive change produced in the abstraction level of the visualization of Luthier subsystems

5. Related Work

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

The use of visualization and animation techniques to assist object-oriented program understanding is specially being 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 specially suitable for program animation tools because they support the definition of events at any level of abstraction, but they are not so adequate to support the analysis of abstractions that require program static information.

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.

The use of 3D visualizations was addressed by Vion-Dury and Santana [VIO 94]. 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.

DePauw, 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* 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 make explicit how design patterns are automatically recognized, or even if actually they are.

6. Conclusions

The understanding of object-oriented programs and particularly object-oriented frameworks is undoubtedly a difficult task. To alleviate this task, MetaExplorer attempts to provide an adequate set of tools that can be used to analyze a framework from different points of view and by users with different background.

The strengths of the reverse engineering approach proposed in this paper was empirically demonstrated through experiments, which suggest that the use of the support tool helps a user to grasp a better understanding of a framework and, therefore, to produce better applications in terms of framework reuse. Particularly, empirical data about the effectiveness of the mechanisms proposed by visualization tools for object-oriented program understanding are rarely found in the literature.

The mechanisms introduced by the Luthier framework allow to implement different visualization tools with little effort. The use of meta-objects-based techniques centered on the concept of meta-object managers enables the construction of sophisticated meta-architectures, specially adapted to the requirements of each tool, in a simple and clear manner. The concepts of abstractor objects and symbolic abstraction scales enable the construction by composition of complex filtering mechanisms that greatly simply the implementation of visualizations, which is often the more time-consuming task in the development of visualization systems. Currently, LuthierAbstractors is demonstrating its versatility to support other types of complex visualizations, as for example, visualizations in geographic information systems.

7. 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.

[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 Engeneering, São Carlos, Brazil. October 1996. (in portuguese)

[CAM 97] Campo, M. *Visual Understanding of Frameworks through Instrospection of Examples*. Ph.D. Thesis. UFRGS, CPGCC, 1997. (in portuguese)

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

[DEP 93] De Pauw, W.; et al. *Visualizing the Behavior of Object-Oriented Programms*. 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, Italia. 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.

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

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

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

[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.*. Englewood Cliffs: Prentice-Hall, 1994.

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

[MUT 95] Muthukumarasamy, J.; Stasko, J. *Visualizing Program Executions on Large Data Sets using Semantic Zooming.* Georgia, Georgia Institute of Technology, 1995. (Tech. Report. GIT-GVU-95-02).

[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.

[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.