Enhancing Framework Usability through Smart Documentation

Alvaro Ortigosa[†*] Marcelo Campo[*] Roberto Moriyón Salomón[†]

*Universidad Nacional del Centro Prov. Bs. As. - Fac. Ciencias Exactas
ISISTAN Research Institute - Objects and Visualization Group,
Campus Universitario, Paraje Arroyo Seco, (7000) Tandil, Bs. As., Argentina
[†]Universidad Autónoma de Madrid, E.S.T. de Ingeniería Informática
Campus Cantoblanco, 28049, Madrid, España
E-mail: {alvaro.ortigosa; roberto.moriyón}@ii.uam.es, mcampo@exa.unicen.edu.ar

Abstract. In this work we present SmartBooks, a new approach to support framework instantiation based on the active cookbook concept, extended with the combination of the concept of user-tasks modeling and least commitment planning methods. Based on this technique, a tool can present to the developer the different high level activities that can be carried out when creating a new application from a framework, taking as basis the documentation provided by the designer through instantiation rules. For each of these high level activities, there is a list of tasks that the user must carry out in order to complete the activity. When the user selects her next objective, the tool is able to build the sequence of tasks that have to be done to accomplish that objective: the instantiation plan; and the process of plan creation is named planning. In this paper we present the main characteristics of the planning approach and an example of the instantiation tool being developed.

1. Introduction

Good quality documentation is an essential requirement to successfully carry out all the tasks involved in software development. Documents flows through the development process to communicate decisions involved in the different stages of the development. Also, documentation plays a central role in software reuse, in which a user must comprehend a piece of software to be reused to build a new application.

This aspect gets a critical relevance in the case of object-oriented application frameworks. Object-oriented application frameworks constitute a great improvement in software reuse because they promote the reuse of not only single building blocks, but also the reuse of the design of systems or subsystems. However, depending on framework complexity, the development of new applications reusing that framework usually is a hard and time-consuming task for novice users [3, 4, 8, 10, 24]. This aspect represents one of the most limiting factors of the technology.

Due to this reason, much effort has been spent during the last ten years to create more powerful documentation techniques. However, traditional design and code documentation techniques are not enough to describe the complexity of a framework, specially if considered the different kind of users that may need to access framework documentation [10]. Butler et al. [4] describe four kind of framework (re)users: application developer, framework maintainer, developer of another framework and verifier. Taking into account this variety, different documentation methods have been proposed for documenting frameworks. Some of them are informal and prescriptive [10, 20, 23], that is, they describe how the framework should be used. Some other methods are more formal and descriptive: they describe the framework design, and the user has to deduce how to use it [8, 24]. Every technique is oriented to a given kind of framework user, and although some of these approaches are able to provide good descriptions of some framework aspects, none of them can successfully satisfy all the framework documentation requirement.

This limitation naturally leads to think on tools that effectively help the user in the instantiation of a given framework as an important complement. In this sense, among the different proposals existing in the literature, the so-called active cookbooks [23, 18] represent one of the most prominent examples of tools that provide semi-automated assistance to the framework instantiation process. Active cookbooks are able to enact recipe descriptions, providing the user an interactive interface that guide her through the instantiation process. Recipes do not explain the design rationale, they just explain how the problem can be solved using the framework. This kind of help facilitates the instantiation of predicted functionality because humans are good at following step-by-step directions, but, paradoxically, its little flexibility represents one of the fundamental drawbacks of the approach. When dealing with an active cookbook the user usually has to follow the embedded recipes up to the last detail, or resign to not using the tool at all.

In this work we present *SmartBooks*, a new approach to support framework instantiation tools based on the concept of active cookbook extended with the combination of the concept of user-tasks modeling and least commitment planning methods. Based on this technique, a tool can present to the developer the different functionalities that can be achieved using the framework, taking as basis the documentation provided by the designer through *instantiation rules*. When the user selects the desired functionality, the tool is able to build the sequence of tasks that have to be done to implement such functionality..

The paper is organized as follows. The next section surveys the state of the art in framework documentation techniques. Section 3 presents the rule-based approach proposed in this work. In section 4 a detailed example of smart instantiation is presented. Finally, in section 5, the main conclusions of the work are presented.

2. A Survey of Framework-Related Documentation Techniques

The problem of framework usability has led to the proposal of multiple documentation approaches aiming to provide a better support for different kinds of framework users. The different techniques proposed during the last years can be historically divided into four main categories: early techniques, framework-specific techniques, electronic-based mixed techniques and tool-oriented techniques.

2.1. Early techniques

The first and simpler technique used for framework documentation is to include with the framework distribution the source code of example applications that have been constructed using the framework. This is often the only documentation provided to application developers, and ideally it should consist of a set of training examples that introduce the framework gradually and illustrate in each step a single new hotspot, starting with the simplest and most common form of reuse for that hotspot. In spite of this, example applications usually have no organization, and the user should browse through the set of classes, looking for an example with similar functionality to the application being built.

Besides example applications, first frameworks where documented using general notations for object oriented software documentation, like for example OMT [22] and OOSE [9]. Because these notations were not designed specifically for frameworks, they fail to capture essential aspects of frameworks, notably collaborations among abstract classes. Moreover, they are oriented toward documenting a given design, but do not provide the support needed for reuse purposes, i.e., how to create a new application from the design.

Another generic technique adapted to frameworks are reference manuals. A reference manual for an object-oriented system consists of a description of each class (responsibility, data members and methods), global variables, constants and types. Applied to frameworks, descriptions should include information about whether a class is intended to be subclassed or a method to be overridden. Reference manuals by themselves are not very useful way to learn a framework [Butler99].

2.2. Framework-specific techniques

One of the first techniques designed specifically for documenting frameworks were recipes and cookbooks [19]. A recipe describes how to perform a typical example of reuse during application development, while a cookbook is a collection of recipes. Recipes do not explain the design rationale, but they just explain how the problem can be solved using the framework. A guide to the contents of the recipes is generally provided, either as a table of contents, or by the first recipe acting as an overview of the cookbook. Examples of this type of documentation are [1] and [12].

In order to provide better structure to cookbooks, Johnson [10] introduced an informal pattern language for documenting frameworks using natural language. Johnson’s patterns provide a format for each recipe, and an organization for the cookbook. The organization follows a spiral approach where recipes for the most frequent forms of reuse are presented early, and where concepts and details are delayed as long as possible, being the first recipe an overview of the framework concepts and the other recipes.

A concept similar to cookbooks but with different goals are the design books [19]. A design book (or design notebook) collects design information, including background theory, domain information, and a discussion of engineering trade-offs. Information about system requirements, specifications, architecture, components design, code and design history is included in the design books. While not specifically intended for frameworks, design books have been adopted [2] to capture the design rationale of software systems, as well as combined hardware/software systems.

As the size and complexity of the frameworks increased, more formal techniques were needed to represent frameworks structures. An example of formal documentation are *interface* contracts [16], which are specifications of obligations, each one providing specification of a class interface and invariant in isolation; an interface contract specifies the type constraints given by the signature of a method, and the interface semantics of the method. Similarly to other techniques focused on individual classes, this approach does not scale up well to frameworks.

On the other hand, *interaction* contracts [8] are also specification of obligations, but they deal with the co-operative behavior of several participants that interact to achieve a joint goal. An interaction contract specifies a set of communicating participants and their contractual obligations: the type constraints given by the signature of a method, the interface semantics of the method, and constraints on behavior that capture the behavioral dependencies between objects. Besides, it specifies preconditions on participants required to establish the contract and the invariant to be maintained by these participants. One of the problems of these contracts is that they can not be used to assist in the building of new applications using the framework. Moreover, the complexity of resulting specifications makes them more adequate for mechanical interpretation.

Design Patterns [6] follow the line of textual patterns, and were created seeking the systematization of the design knowledge and experience. They capture design experience at the micro-architecture level, by specifying the relationship between classes and objects involved in a particular design problem; a design pattern presents a solution to the problem and provides an abstraction above the level of classes and objects. Describing the pattern in an abstract level enables to reuse it for the design of new applications, as well as to improve existing application designs, independently of implementations. One of the goals of design patterns is to provide meta-knowledge about how to incorporate flexibility into a framework. Besides, this kind of information can be very useful for documenting the framework design in a higher abstraction level, as they are good at describing architectures. The description of the design pattern explains the problem and its context, the solution, and a discussion of the consequences of adopting the solution, and is generally illustrated by a concrete example. The solution describes the objects and classes that participate in the design, and their responsibilities and collaborations, generally using a collaboration diagram and providing examples of the pattern application to concrete situations. Benefits and drawbacks of applying the pattern are also discussed.

2.3. Electronic-document-based mixed techniques

Several works have proposed combinations of two or more of the described techniques, using hypertext to link the different documentation components. For example, Lajoie and Keller [13] extended Johnson’s textual patterns, introducing the term *motif* to name these patterns, in order to avoid confusion with design patterns. They use a template for a motif description that has a name and intent, a description of the reuse situation, the steps involved in customization, and cross references to motifs, design patterns and contracts. The design patterns provide information about the internal architecture, and the contracts provide more rigorous descriptions of the collaborations relevant to the motif.

A similar approach is proposed by Demeyer et al. [5], with the intention of assuring consistency between framework implementation and documentation. The approach automates the maintenance of the hypermedia links between framework documentation and framework source code of on-line documentation by using computed hypermedia links, that is, links that perform a computation to determine the target of a navigation action.

Finally, Meusel et al.[15] propose a model for structuring different documentation techniques into a single structure, based on the pyramid principle: the documentation follows a top-down structure, with hyperlinks relating information at the same or contiguous level of abstraction.

Recently there has been a growing effort toward providing more formality to the diverse documentation techniques. The goal is to provide less ambiguous communication among experts, as well as to enable CASE tool support for the design and instantiation processes. That is the case of the technique for precise visual specification of design patterns [14]. In this approach, patterns descriptions are separated in three models (role, type and class), and each model is documented using UML [21], extended with recent advances in visual modeling notations to achieve greater precision. Similarly, Soundarajan [24] presents a trace-based approach for specifying the behavior of the framework, in particular the control flow. The author proposes to use this formal specification as a complement to more informal techniques, like the study of existing applications.

2.4. Tool-oriented techniques

Another approach to framework documentation is represented by exemplars [7]. An exemplar is an executable visual model consisting of instances of concrete classes together with explicit representation of their collaborations. For each abstract class of the framework, at least one of its concrete subclasses is instantiated in the exemplar. This technique is oriented towards providing assistance to the framework instantiation process, that is, the process of creating a new application based on the framework. The user creates the new application by gradually adapting the exemplars according to the application requirements, being able to visualize in each step the result of modifications. In spite of the usefulness of starting from an existing application, this approach has some drawbacks. The visual models are hard to build, and there are limits to what can be done through them. Besides, this kind of documentation, similarly to cookbooks, does not provide information about the design rationale.

Currently, one of the most promising techniques are active cookbooks [20], which are tools that provide semi-automated assistance to the framework instantiation process. Active cookbooks are able to enact recipe descriptions, providing the user an interactive interface that guides her through the instantiation process. This kind of help facilitates the instantiation of predicted functionality because humans are good at following step-by-step directions, but, paradoxically, its little flexibility represents one of the fundamental drawbacks of the approach. When dealing with an active cookbook the user usually has to follow the embedded recipes up to the last detail, or resign to not using the tool at all.

2.5. Analysis

Analyzing the described approaches can be observed a general agreement about good framework documentation should provide more than a way to represent the framework design. Techniques like exemplars and cookbooks (including active cookbooks) are goods at providing procedural assistance, but the lack of design rationale makes it hard to adapt the documented framework in no anticipated ways. On the other hand, more passive approaches, especially those combining more than one documentation technique, provide more flexibility, at the price of imposing a greater overload on the framework user. This additional effort is more evident with novice users, who need to spend considerable time reading the documentation before being able to build the application. Additional flexibility is needed when the documentation has to assist users with different goals. Mainly three audiences consult framework documentation: users selecting a framework, users developing typical framework-based applications and users intending to modify the framework architecture.

Considering this, the best approach seems to be the use of models that combine active documentation with detailed design explanations, both formal and informal. Even so this is proposed by some of the described works, specially those built around a hypermedia system, none of them describes how this combination can be achieved, beyond the use of hyperlinks to relate the different documentation portions. This is not enough if taking into account the complexity derived from the amount of information involved in documenting a framework. In this way, new approaches that enable to structure and organize the different documentation techniques in more efficient ways are needed.

Also, a higher-level instantiation mechanism is necessary. This mechanism, for example, should allow a user to select the desired functionality of her application and produce directives that guide her to obtain such functionality using the framework. The SmartBooks environment aims to provide this functionality through a rule-based documentation approach as is described in the next section.

3. SmartBooks: Towards Smart Framework Documentation

The SmartBooks environment distinguishes between two different user types: the framework designer (or documentation writer[1]) and the framework user (also called application developer). The framework developer can describe the design using common documentation techniques, i.e. UML diagrams [Rational97], design patterns, examples, etc. One important difference with other approaches is that, besides this documentation, the environment enables the designer to describe the functionality provided by the framework, how this functionality is implemented by different framework components, and to provide rules that constraint framework specialization.

[1] In the rest of the paper it is used the term “framework designer” (or just designer) to name the responsible for writing the framework documentation. Even so this is not necessarily the case (the framework can be documented by documentation specialists), the assumption does not imply loss of generality.

Using this documentation, the system provides information that guides the framework user through the process of application development. This guidance is focused on the intended functionality for the new application, so the user is oriented to define what the application is supposed to do. Based on the information about application functionality, a plan for the framework instantiation is elaborated, and the user must accomplish the task list that composes the plan. Each generated pending task is related to the corresponding documentation. That is, when developing a given task, the user is able to navigate to related documents, as for example, class hierarchy, collaboration diagrams, design patterns, examples, rules, associated code, etc.

SmartBooks is based on the idea that a tool should incorporate the notion of *instantiation tasks*. That is, considering that the instantiation of software frameworks is an activity based on a well-defined amount of basic tasks, like for example class specialization and method overwriting, such activities can be assimilated to the concept of user-tasks, successfully used in the modeling of interactive applications [11, 17]. In this way, the process of producing instantiation documentation can be seen as a process of user-tasks modeling.

Generally, this modeling consist of the definition of a hierarchy of tasks, in such a way that lower level tasks correspond to the semantic interpretation of the most elementary user actions, while higher level tasks are the concatenation of others with a lower level. Modeling user tasks in complex applications allows a richer interaction between the user and the system. Thanks to this, the system is able to understand with higher precision the objectives of the user, thus being able to assist her in their consecution.

In the case of framework instantiation, a task manager has to allow tasks to be achieved in any order, to be interrupted in order to work on other tasks, and even to be cancelled at any moment, and the system must be able to adapt itself to the new situation risen at each of these steps. In this sense, the use of least commitment planning techniques [25] appears as an interesting alternative to build partial plan sequences of instantiation tasks in a flexible way.

In this way, if common framework documentation techniques are enhanced with *instantiation rules* that describe the necessary actions to be taken in order to implement some functionality, the tool would be able to apply a planning algorithm to produce a task sequence to guide the user in the framework adaptation.

3.1 Instantiation Rules

SmartBooks uses a planning algorithm specially developed to produce sequences of instantiation tasks based on a set of functionality goals to be satisfied by the new application. The input of the planner is a set of rules that describe the necessary steps to obtain the desired functionality. Some of these rules can be framework-specific, while others can describe general situations of framework instantiation. The general form of a rule is:

```
        precondition list ← postcondition
```

and represents changes on the software and/or the plan state.

These rules state which preconditions are needed for the post-condition to be true. So, in every step the planner tries to make true the preconditions of an action whose post-conditions are goals.

The following is an example of specific rules for the HotDraw framework[2]:

```
        when(useFigure), tryUseComponent(X,'Constraint')
                  ← functionality("Establish relationships between attributes")
        selectTool(Tool, Goal), addTool(Tool) ← useTool(Goal)
```

The first rule states that if it is necessary to establish relationships between attributes, *Constraint* must be used. Besides, the functionality will be available if *Figures*, other framework components, are being used. The second rule represents that for using a *tool* the tool should be first selected and then added to the application being built.

It must be noticed here that the designer arbitrarily fixes the terms used to express the functionality. In this way the tool must show to the user the different functionality implemented by the framework. This representation can be further refined if a domain language were defined to provide a textual vehicle to express the functionality desired for a specific application, although this aspect is beyond of our current goals.

An example of a generic rule is a rule that states how a component can be used is:

```
exists (class (CompDesc, X)), choose (CompDesc, ['refine', 'reuse'], Answer),
              useComponent (C, CompDesc, Answer)  ← tryUseComponent (C,
CompDesc).
¬ exists (class (CompDesc,X)), defineNewComp (C,CompDesc)
                ← tryUseComponent (C, CompDesc)
```

The first rule states that, in order to use an existing component, the user can choose between use it "as is", or create a specialization of it. The second one shows what to do if the component does not exist. These framework independent rules are called "primitives", because are the building blocks to describe the framework specific rules.

In SmartBooks these rules are automatically generated through a graphical interface that allows the designer express instantiation actions along with general design documentation of the framework. Functionality description is made at different abstraction levels. While some functionality can be directly associated with components or groups of components, some other should be described in terms of lower level functions. An example of the first type of descriptions is the description of the implementation of animated drawings in HotDraw, shown in figure 1. These descriptions not only give orientation about what component implements a given functionality, but also provide hints about how it should be used.

[2] The examples presented it this paper are based on the HotDraw framework [Johnson92]. This framework was selected due to its moderate size, and because it has been used to illustrate other approaches to framework documentation, which facilitates contrasting the techniques. The HotDraw version used for the analysis is version 41.4.

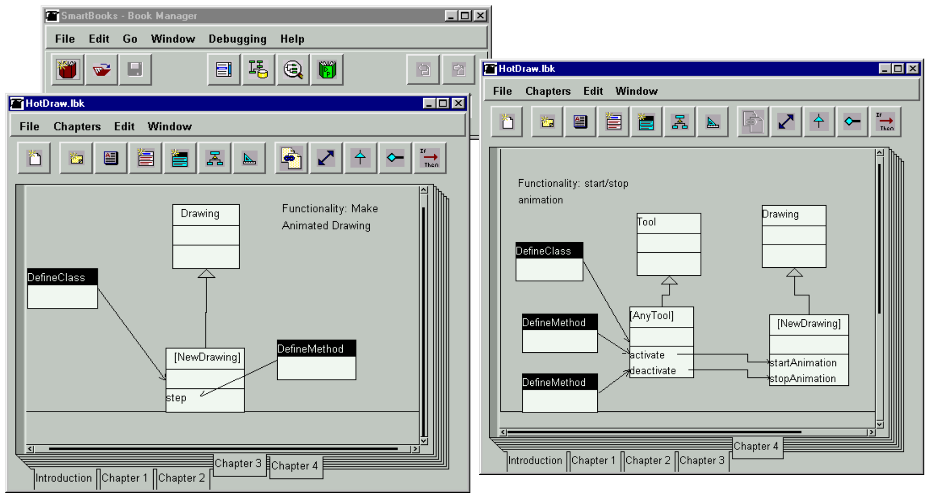


Fig. 1. Instantiation Rule Definition in SmartBooks

The diagram on the left window defines that, in order to implement animated drawings, a subclass of *Drawing* should be created, and the *step* method should be defined for this subclass. More precisely, the diagram prescribes that two tasks should be carried out by the user: a *DefineClass* task, that has to produce a subclass of *Drawing*, and a *DefineMethod* task, which must produce a method with name *step* for the new subclass. Besides, it provides explanations about the expected behavior of the method.

Some primitive rules have, as a side effect, the creation of tasks that must be carried on by the user. For example, if the planner finds that the implementation of a given functionality requires the specialization of a component and the redefinition of some methods, the corresponding tasks are created and queued as "pending" tasks to the user. The following primitive rule, for example, describes how a new component can be instantiated:

```
pendingTask ('DefineNewClass', FuncDescs) ← implementComponent (C,
FuncDescs)
```

There exists a second type of task, called "waiting tasks", which represent tasks whose result is necessary for the planning algorithm to continue. When the planning algorithm finds a waiting task as a precondition of a desired post-condition, it creates the task and is suspended. Once the user completes the task, the algorithm is reinitiated. One of the primitive rules used to get input from the user is

```
waitTask ('GetInputTask', Description, Return) ← getUserInput (Description,Return)
```

The next section presents an example of the use of SmartBooks for producing a framework instantiation.

4. An Example of Smart Guidance through Instantiation Rules

The following rules are an example subset of the ones generated to define the functionality provided by the HotDraw framework and the activities that have to be carried out to add the functionality to a new application, using the Documentation Tool.

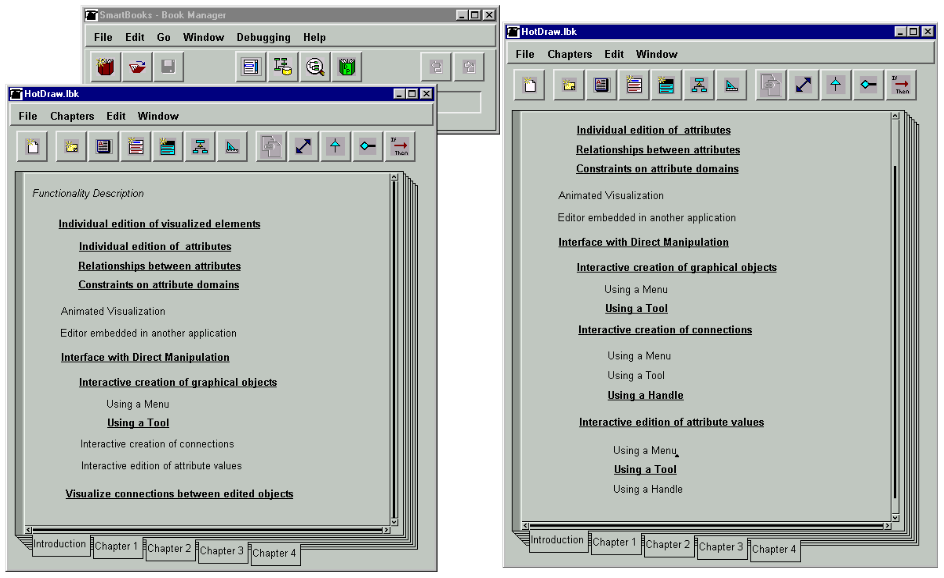
1. addFigure ← initialFunctionality (“Individual edition of visualized objects”)
2. option([editAttribute, createFigure, createConnection]) ← initialFunctionality (“Direct Manipulation”)
3. useComponent(Figure,Description,’refine’) ← addFigure
4. functionality(“Interactive Edition of Attribute Values”), option([useEditionTool(‘editAttribute’), useHandler(‘editAttribute’), useMenu(‘editAttribute’, AFigure)]) ← editAttribute
5. useComponent(CompositeFigure,Description,’refine’), useTool(“editAttributes”) ← useEditionTool
6. selectTool(Tool, Goal), addTool(Tool) ← useTool(Goal)
7. pendingTask(“useExternalTool”,[“ToolBuilder”, Tool]) ← selectTool(Tool, Goal)
8. defineComponent(“Tool”) ← selectTool(Tool, Goal)
9. pendingTask(“UpdateMethod”,[Editor, tools]) ← addTool(Tool)
10. reuseComponent(Component) ← useComponent(Component)
11. defineComponent(Component), documentComponent(Component) ← useComponent(Component)
12. pendingTask(“SelectExistingComponent”, [SelectedComp,Component]) ← reuseComponent(Component)
13. pendingTask(“DefineClass”,[NewClass, Component]), pendingTask(“DocumentClass”, [NewClass]) ← defineComponent(Component)
14. pendingTask(“DefineMethod”, [Method,Component]), pendingTak(“DocumentMethod”, [Method,Class]) ← defineMethod(Method,Class)

The first step when creating a new application is to define the required functionality for the application. The functionality required for a PERT editor, informally described, is:

> *“It should be possible to interactively create graphical objects representing events, and to relate these events through precedence links. The events should have visual representation for its attributes, an two of the attributes will be edited through the graphical interface. Besides, each attribute could be related with other ones, both from the same or related events.”*

At the beginning, the initial available functionality is presented, based on the *initialFunctionality* rules (Fig.2). When the user selects the required functionality, the corresponding rules are activated, and the left part of each rule is added to the goals of the planning algorithm. These rules also include fields used to explain the functionality to the user, but they are not shown here for space reasons.

This functionality is selected by the user, using the SmartBook's Functionality Collector. Figure 2 shows partial views of this description, which is made gradually. At first (left window), only the highest level functionality is presented to the user. As the user selects the functionality, if available, options are displayed, so she can refine her selection. For example, when the user selects to incorporate Direct Manipulation functionality to the interface, the options to create graphical objects, create relationships, and edit attributes are presented together with some other alternatives. The window at the right of figure 2 shows a subsequently stage on the functionality description.



Functionality Description

Individual edition of visualized elements

Individual edition of attributes

Relationships between attributes

Constraints on attribute domains

Animated Visualization

Editor embedded in another application

Interface with Direct Manipulation

Interactive creation of graphical objects

Using a Menu

Using a Tool

Interactive creation of connections

Interactive edition of attribute values

Visualize connections between edited objects

Individual edition of attributes

Relationships between attributes

Constraints on attribute domains

Animated Visualization

Editor embedded in another application

Interface with Direct Manipulation

Interactive creation of graphical objects

Using a Menu

Using a Tool

Interactive creation of connections

Using a Menu

Using a Tool

Using a Handle

Interactive edition of attribute values

Using a Menu

Using a Tool

Using a Handle

Fig. 2. Selecting the application functionality. Underlined items represent the user selections

In the example, the user selects "*Individual edition of visualized objects*" and "*Direct Manipulation*". From the rule 1, the *addFigure* goal is transferred to the planning algorithm. According to the rule 3, this goal has a subgoal, *useComponent(Figure,Description,'refine')*. Following the other selected functionality, "Direct Manipulation", rule 2 is activated, which allows the user to refine the functionality selection. Once the user has selected the suboption, in this case "Interactive Edition of Attribute Values" (rule 4), the planning process continues. Rule 4 presents another alternative, so the user is asked to select the way the attributes will be edited. If the user selects to use an edition tool, rule 5 is used to determine how to achieve the goal. This rule introduces two new goals: to use a *CompositeFigure* and to use a *tool*.

At this point, a special feature of the planning algorithm can be described. From two different requirements there have been generated *the useComponent(Figure, Description, 'refine')* and *useComponent(ComposedFigure, Description, 'refine')* goals. As *ComposedFigure* is a subclass of *Figure*, the second goal is actually a refinement of the first one. In that way, every time the system detects this type of situations, it merges the two goals, preserving the most specific one. In this case, the system continues working with goal related with *ComposedFigure*.

Trying to satisfy the new *useComponent(ComposedFigure,...)* goal, two different plans can be generated: reusing an existing *ComposedFigure* subclass or the defining a new subclass (rules 10 and 11). The first plan implies creating a task to select the component to be reused; this task is created and inserted in the list of pending tasks. If the user rejects this alternative (that is, if she rejects the proposed task), the planning system tries the second option, and creates two pending tasks for defining and documenting a new *ComposedFigure* subclass (rule 13). In case the designer wants the user to make a choice among the possible plans for a given goal, an *option* plan can be provided as we have seen in

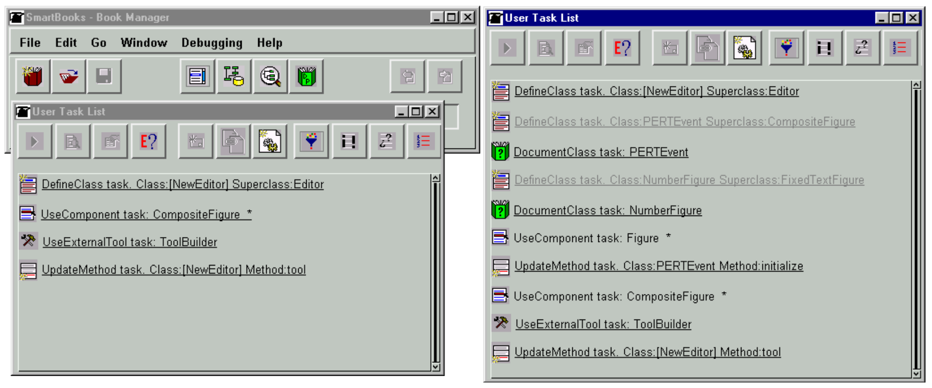
Still remains a goal to be satisfied, namely *useTool("editAttributes")*. Rule 6 states that resolving this goal implies to selecting a tool and adding the tool to the editor being implemented. Selecting a tool, has two alternatives: rules 7 and 8. Again, in a first stage the planning algorithm chooses the first rule, generating a task to use an external tool, the *ToolBuilder* provided with HotDraw. If the user prefers it, the second alternative can be adopted, creating a task to define a new component,a subclass of *Tool*.

Finally, to achieve the second goal generated from rule 6, a task is generated to update the *tools* method of the *Editor* class.

At this point, the list of tasks to be executed by the user is the following:

DefineClass (NewClass, "ComposedFigure")
DocumentClass (NewClass)
UseExternalTool("ToolBuilder", NewTool)
UpdateMethod(Editor, "tools")

The instantiation plan is presented to the user by the Task Manager as a list of pending tasks. Figure 3 shows two lists of tasks for the instantiation of a Pert Editor, corresponding to two different instants of the instantiation process. The list on the left shows an early moment of the process: create a subclass of *Editor* (required for every application built using *HotDraw*), define the class of *ComplexFigure* that will used to represent the *Events* (the symbol * represents that this task can be executed more than once, if additional complex figures are required), select a tool for creating the *Events* and add this tool to the editor. Through the Task Manager interface, the user can see both the executed and the pending tasks. She can also navigate to the documentation related with a given task, to select the next task to be executed, to undo or cancel a task.



DefineClass task. Class:[NewEditor] Superclass:Editor
UseComponent task: CompositeFigure *
UseExternalTool task: ToolBuilder
UpdateMethod task. Class:[NewEditor] Method:tool

DefineClass task. Class:[NewEditor] Superclass:Editor
DefineClass task. Class:PERTEvent Superclass:CompositeFigure
DocumentClass task: PERTEvent
DefineClass task. Class:NumberFigure Superclass:FixedTextFigure
DocumentClass task: NumberFigure
UseComponent task: Figure *
UpdateMethod task. Class:PERTEvent Method:initialize
UseComponent task: CompositeFigure *
UseExternalTool task: ToolBuilder
UpdateMethod task. Class:[NewEditor] Method:tool

Fig. 3. List of Pending Tasks. Underlined task are required, while the other ones are optional. The tasks shown in gray represent tasks that were already executed

The list on the right shows the state of the plan after the user has executed the *DefineClass* task (creating the class *PERTEvent*), and has created a class for visualizing its attributes (*NumberFigure*).

Another task is created and associated with the new class; this task will guide the user on the documentation of the class, that is, to describe the functionality implemented by it. This information will enable the reuse of the component on future developments, if the implemented functionality is needed again. These *documentation tasks* are generated every time the user creates a new class or method, and the associated functionality can not be deduced from the instantiation process. Nevertheless, the user can remove the task without executing it (actually, the user can reject any task created by the system), if the documentation is not considered relevant.

Even so the system provides forms for executing basic tasks, the user can use other tools to manipulate the code. In the current Smalltalk 80 implementation of SmartBooks, regular Class Browsers can be used to work on the code. Every operation on the code made by the user is translated to the equivalent task, and the system tries to associate the user action with some task waiting to be executed. Whether it corresponds to a pending task or not, the task is put on the list of executed tasks.

SmartBooks allows the user to redefine the application requirements either while instantiating the framework or when the application development has finished. For example, if the user decides to modify the way *PERTEvent* attribute values are changed, using menus instead of tools, the system will generate a new instantiation plan. In the new plan, the tasks for selecting a tool and adding it to the editor will be replaced by a task to modify the *menu* method of the *PERTEvent* class. All the tasks of the new plan that were already executed as part of the old plan, are considered as executed. Similarly to any executed task, the user can use the result or undo any of the tasks.

5. Conclusions

In this paper, a new approach for supporting framework instantiation, called SmartBooks, was presented. SmartBooks extends the active cookbook concept, by combining user-task modeling with least commitment planning.

The main contribution of our work is to show how the instantiation process can be represented as a sequence of user-tasks, which are derived by a planning algorithm from the application functional description. This plan can be dynamically updated accordingly to the developer actions or when the initial conditions (that is, functional requirements and framework documentation) are modified. With this goal, the framework is documented using rules to describe how the functionality is implemented by the framework components, and to express restrictions on the way the components can be used and specialized.

One of the limitations of the proposed approach is the necessity of describing the functionality implemented by the framework using natural language. Vocabulary, assumptions and visions of the world can have big variations from one user to another, and especially with reference to the designer background. Because of this, expressing the functionality and understanding the framework capabilities are hard tasks imposed on the designer and users, respectively. In order to reduce the difficulty, framework designers can currently associate longer explanations and examples with each functionality item. Nevertheless, the problem must be carefully studied so to find a less ambiguous way of describing functionality, without the difficulties of using some formal notation.

6. Acknowledgments

The authors thanks Analía Amandi for the invaluable discussions about this work and comments on the manuscript.

7. References

1. Adobe Systems. Postscript Language - Tutorial and Cookbook. Addison-Wesley, 1985
2. G.Arango, E.Shoen, R.Pettengill. A process for consolidating and reusing design knowledge. Proceedings of 15th International Conference on Software Engineering, IEEE Computer Press, 1993.
3. Butler G., Keller R., Mili H. A Framework for Framework Documentation. ACM Computing Surveys, special Symposium Issue on Object-Oriented Application Frameworks. To appear.
4. Butler G., Dénommée P. Documenting Frameworks to Assist Application Developers. In Object-Oriented Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, 1999.
5. Demeyer S., De Hondt K., Steyaert P. Consistent Framework Documentation with Computed Links and Frameworks Contracts – ACM Computing Surveys – Symposium on OO Application Frameworks. 1998.
6. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addisson-Wesley, Reading, Mass., 1994.
7. D.Gangopadhyay, S.Mitra - Understanding Frameworks by Exploration of Exemplars. Proceedings of 7th. International Workshop on C.A.S.E. (CASE-95). IEEE Computer Society, July 95.
8. R.Helm, I.M.Holland,D.Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In Proceedings of OOPSLA'90, ACM/SIGPLAN. New York, 1990.
9. Jacobson I., Christerson M., Jonsson P., Overgaard G. Object Oriented Software Engineering: A Use Case Driven Approach, ACM Press, 1992.
10. Johnson R. Documenting frameworks using patterns. In Proceedings of OOPSLA'92. ACM/SIGPLAN, New York, 1992.
11. Johnson P., Wilson S., Markopoulos P., Pycock J.. ADEPT, Advanced Design Environment for Prototyping with Task Models. Proceedings of INTERCHI'93, ACM Press.
12. G.E.Krasner, S.T.Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming, 1, 3 (1988).
13. Lajoie R., Keller R. Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert. Proceedings of the 62nd Congress of the ACFAS, Canada, May 1994.

14. Lauder A., Kent S. Precise Visual Specification of Desing Patterns. Proceedings of ECOOP 98. Lecture Notes in Computer Science, Springer Verlag, 1998.
15. Meusel M, Czarnecki K., Kopf W. A Model for Structuring User Documentation of Object-Oriented Frameworks Using Patterns and Hypertext. Proceedings of ECOOP'97 – Object Oriented Programming. Lecture Notes in Computer Science, Springer Verlag, 1997.
16. Meyer B. Applying design by contract. IEEE Computer, October 1992
17. Paternò, F. Understanding Task Model and User Interface Architecture Relationships, CNUCE Internal Report, December 1997.
18. Pree W. Framework Development and Reuse Support. In Visual Object-Oriented Programming, Concepts and Environments. M.Burnett, A.Goldberg, T.Lewis (eds.). Manning - Prentice Hall. 1995.
19. Pree W. Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995.
20. Pree W., Pomberger G., Schappert A., Sommerlad P. Active Guidance of Framework Development. Sofware-Concepts and Tools. Springer-Verlang, 1995.
21. UML Semantics, version 1.1 September 1997. http://www.rational.com/uml
22. Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W. Object-Oriented Modeling and Design. Prentice Hall, 1991.
23. Schappert A., Sommerland P., Pree W. Automated framework development. Symposium on Software Reusability (SSR'95), ACM Software Engineering Notes. Aug. 1995.
24. Soundarajan N. Understanding Frameworks. In Object-Oriented Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (Eds.) John Wiley and Sons, N.Y, 1999.
25. Weld D. Recent Advances in AI Planning. AI Magazine, 1999.