Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 74 (2005) 55–64
The Journal of Systems and Software
www.elsevier.com/locate/jss

"Computer, please, tell me what I have to do...": an approach to agent-aided application composition

Marcelo R. Campo *, J. Andrés Díaz Pace, Federico U. Trilnik

ISISTAN Research Institute, Faculty of Sciences, UNICEN University, Campus Universitario, Paraje Arroyo Seco, (B7001BBO) Tandil, Buenos Aires, Argentina

CONICET, Avda. Rivadavia 1917, (C1033AAJ) City of Buenos Aires, Argentina

Received 16 October 2002; received in revised form 1 February 2003; accepted 2 May 2003
Available online 21 January 2004

Abstract

The process of starting to use any reuse technology is usually one of the most frustrating factors for novice users. For this reason, tools able to reduce the learning curve are valuable to augment the potential of the technology to rapidly build new applications. In this work, we present *Hint*, an environment for assisting the instantiation of Java applications based on software agents technology. *Hint* is built around a software agent that has the knowledge about how to use a reusable asset and, using this knowledge, is able to propose a sequence of programming activities that should be carried out in order to implement a new application satisfying the functionality the user wants to implement. The most relevant contribution of this work is the use of planning techniques to guide the execution of instantiation activities for a given technology.

1. Introduction

It is a well-known fact that the more powerful the reuse technology the more knowledge is necessary to rapidly start to use it to produce applications. This aspect represents one of the most limiting factors of any reuse technology, but it is particularly crucial to object-based ones (Bosch, 2000). For this reason, composition tools are an invaluable complement. These tools can vary from simple wizards for generating code skeletons to complex graphical tools supporting the visual composition of applications. This kind of tools can dramatically improve the productivity in the case of applications that naturally fit into the scope of the target technology (Campo et al., 2002). However, when complexity grows, despite components are derived from a domain-specific framework, an integration framework or they are implemented following some interface standard, some kind of coding is always necessary in order to get a running application.

At this point a deeper knowledge of underlying design details can be necessary in order to use, or even more, adapt the behavior of existing components. Certainly, good quality documentation is a key issue. Nevertheless, none of the developed documentation techniques (Demeyer et al., 2000; Johnson, 1992) can be completely adapted to the different types of users, especially if considering the variations on knowledge and experience these users usually have (Helm et al., 1990). On one side, expert users may prefer to know about design details, and be able to make their own decisions. Many times this kind of users can adapt a framework (Fayad et al., 2000) in unexpected ways. On the other side, and perhaps the most important one, novice users may just want to be aware of higher-level aspects. This kind of users should be able to build an application without the need of understanding overwhelming details of design rationale. However, this is not always the case, producing a negative impact on the benefits that the technology can bring to enhance software development.

We believe that the problem resides not in how to provide a specific tool for a given technology, but in how

* Corresponding author. Tel.: +54-2293440363; fax: +54-2293440362.
E-mail addresses: mcampo@exa.unicen.edu.ar (M.R. Campo), adiaz@exa.unicen.edu.ar (J.A. Díaz Pace), ftrilnik@exa.unicen.edu.ar (F.U. Trilnik).

0164-1212/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2003.05.003

to make the documentation an active source able to guide a user in what should be done for building a specific application. On the basis of this documentation, a tool should take the principal requirements and design decisions from the user, and then propose a number of actions to follow in order to get a running application. This process can be approximated by means of what we have called the "Computer, please, tell me what I have to do..." paradigm. The rest of the paper presents an in depth description of this agent-based approach, organized as follows. Section 2 introduces the main concepts of the mentioned paradigm. Section 3 outlines the architecture of the *Hint* environment, illustrated with an example of framework instantiation. Section 4 covers the *Smartbooks* documentation method through which the instantiation knowledge is described. Section 5 shortly explains the use of planning capabilities to elaborate instantiation plans. Section 6 discusses preliminary results and lessons learned. Finally, Section 7 rounds up the conclusions of the work.

2. The "Computer, please, tell me what I have to do..." paradigm

Certainly, a pursued *ideal* is to have a system that (like in *Star Trek*) we could ask the computer to solve any problem by simply asking: "Computer, please, solve this problem...", and automatically obtain the result without any further effort. For example, let's suppose we want to build a graphical editor for Pert-like charts and there is a framework for building graphical editors available. In the ideal situation we would ask the computer to build a system that should satisfy the following informal specification:

> *"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, and 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".*

The computer would interpret these requirements and using the existing framework and components would produce the required application. Unfortunately, computer science is currently rather far of providing such a system, particularly in domains such as software development.

Despite this reality, we can approximate this ideal by using a paradigm in which the computer tells us what steps we should carry out in order to get an implementation using a given reuse technology. That is, given a set of functional requirements for an application, we should be able to ask the computer, "Computer, please, tell me what I have to do to implement this functionality using the existing software that you know how to use". In this approach, instead of providing the computer with the previous requirements, the computer shows the user the different functionality that could be derived from an existing reusable asset and the user can select the aspects he/she judges related with the required functionality. Next, using the knowledge about the existing assets, the computer answers the list of programming activities that should be carried out in order to implement such application. These activities can vary from the advice of which classes should be specialized, what methods should be overridden, what component should be used, what proxies should be implemented, what parameters should be set with specific values, etc. Note that an important aspect of the approach is the interaction with the user, so that he/she can provide the tool with the main design decisions to guide the generation of programming activities.

Following this idea, we developed *Hint*, *a Java* tool based on agent technology (Bradshaw, 1997) designed to provide semi-automated support to the process of application composition. *Hint* is based on the *Smartbooks* documentation method (Ortigosa et al., 2000), which extends common documentation techniques with *instantiation schemes* specifying how a piece of software should be specialized or used to implement a given functionality. Using *least-commitment planning techniques* (Weld, 1994), the *Hint* agent is able to build an implementation plan from the set of functionality that the user selected from the possible functionality that the documented reusable asset can provide. This plan consists of the sequence of programming tasks to be accomplished in order to produce a final application. When the developer starts executing these tasks, the agent observes the process and proceeds to modify the instantiation plan whenever new information, not previously available, can be deduced from the developer's behavior. These features distinguish *Hint* from other approaches in that developers, particularly novice ones, can start the development with a guide of what steps they have to carry out in order to build their applications.

3. The *Hint* environment

Hint represents the result of a three-year research effort on the subject (Ortigosa et al., 2000), incorporating in its last release enhanced functionality to deal with Java frameworks and components, CORBA adaptation and aspect-oriented development (Kiczales et al., 1997). In the current version it comprises four main components: *Documentation Tool*, *Rule Generator*, *Functionality Collector*, and the *Hint* agent, which is in turn composed by three components: *Planner*, *Task Manager*

and *Consistency Manager*. All these components are implemented in Java. Fig. 1 shows these components and their interactions. A short description of each module is included below.

- *Documentation Tool:* It is mainly used by the designer to write the documentation of the reusable assets. This documentation must define the instantiation schemes that describe how to use the assets to derive different functionality. Besides, it is used by the application developer when creating new components or adding functionality to existing ones.
- *Rule Generator:* This component takes the documentation of instantiation schemes as input, and then generates a rule-based representation to be used by the *Hint* agent in the generation of the instantiation plan.
- *Functionality Collector:* This module helps the user to describe the functionality required for the application currently being developed. With this purpose, it uses the information provided through the instantiation schemes in terms of functionality that can be derived.
- *Hint Agent:* The agent is composed by three main functional components, namely: *Planner*, *Task Manager* and *Consistency Manager*.
  - *Planner:* The planner component represents the core component of the *Hint* environment. Based on the requirement information and the schemes, it produces a partial plan for creating the application. The plan is generated using a specially developed planning algorithm, called *PHint*, which is an adaptation of the UCPOP algorithm (Weld, 1994) to support partial and incremental planning.
  - *Task Manager:* It controls the activity of the application developer. It manages the list of pending tasks and informs the *Consistency Manager* when a task is finished, so it can check if consistency rules should be applied. Additionally, it is responsible for enabling the reuse of executed tasks if the instantiation plan is regenerated.
  - *Consistency Manager:* Its work is much related with the *Task Manager*. The *Consistency Manager* has the responsibility of verifying if some user actions produce a software configuration inconsistent with the design description. In such a case, it creates the tasks the user should execute to return to a consistent configuration, and then passes these tasks to the *Task Manager*.

In order to make simpler the understanding of the approach, the next subsection introduces an example of framework instantiation for aspect-based applications using *Hint* with the *Aspect-Moderator* framework (AMF) (Constantinides et al., 2000).

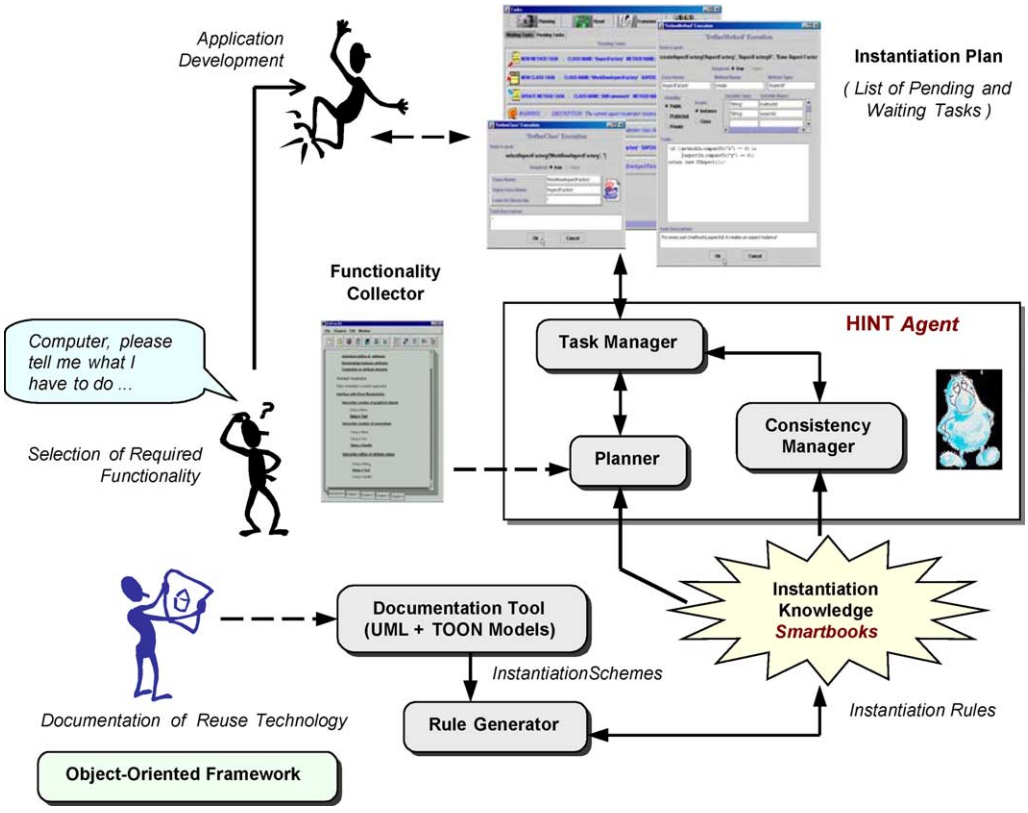

Fig. 1. Main components of the *Hint* architecture.

3.1. An instantiation example

Let's take a simple workflow model for documents. The functionality required for this example can be informally described as follows:

> *"The application is a workflow model for documents with users playing different roles (e.g., manager, normal user or auditor) accessing these documents. Each role has set its own permissions to operate on these documents. A manager can read, write or check documents; a common user is only allowed to read or write some documents; and an auditor can read or check other documents. These capabilities require the application to consider authentication and logging issues. Besides, all these activities can take place simultaneously on the same document so that some concurrency protocols are also needed."*

For this workflow case-study, we have identified the aspects given in Table 1.

With these requirements in mind, we want to implement an application using the aspect facilities provided by the AMF framework. This case-study was actually implemented using the *Smartweaver* environment (Diaz Pace et al., 2002), an extension of the *Smartbooks* method to deal with aspect-oriented development.

When running the *Hint's* Functionality Collector, it starts presenting the user the initial available functionality, based on the instantiation knowledge previously provided by the designer. First, only high-level functionality is presented to the user. As some items are selected, other options including further information may be displayed so he/she can refine previous selections. Fig. 2 shows a sample of specific functionality items relevant to any aspect-oriented development. For example, if we want to specify the cross-cutting of a given aspect with document instances, there is a number of alternatives to choose, namely: a before advice, an after advice or a before/after advice.

On the basis of the requirements collected by the Functionality Collector, the tool internally generates a set of goals to express these requirements and the planning agent responds accordingly with an instantiation (Ortigosa and Campo, 2000). The instantiation plan is suggested to the user by the Task Manager, in terms of *waiting tasks* (representing design decisions) and a list of *pending tasks* (representing instantiation actions).

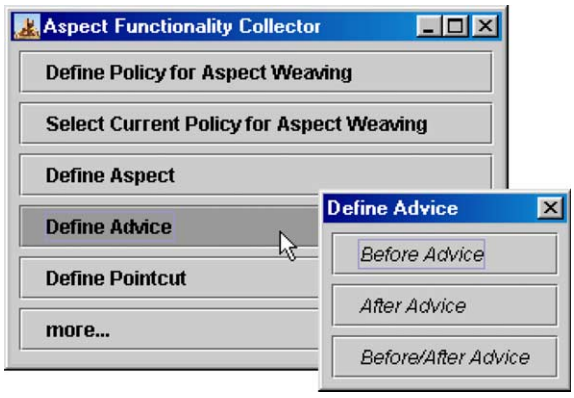

Fig. 2. Functionality items suggested by the *Hint* Functionality Collector.

As an example of this guidance, let's consider some of the activities derived from the definition a new aspect. In the AMF, this functionality involves the definition of an *AspectFactory* class. Hence, the planner asks the user to provide the name for this class. If the class already exists, the planner may initially check for a sub-classing relationship, and then present a task for selecting what methods are to be overwritten in the corresponding subclass. In addition, a task for creating the target subclass may be generated during this process. Fig. 3 shows some of the tasks presented by the Task Manager interface, as a result of the selected functionality. In general, through this interface, the user can see both executed and pending tasks. He can also navigate the documentation associated with a given task, select a given task to be executed, undo or cancel tasks.

Besides, other tasks called *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. These tasks will guide the user on the documentation of the component (the description of its functionality). This information

Table 1
List of some programming tasks supported by *Smartbooks*

Aspect	Description
Concurrency	It mainly deals with synchronization and scheduling issues, by controlling the access to a given document of many potential readings or writings coming from different user sessions. It should consider mechanisms to lock/unlock documents and mechanisms to determine who is the next reader/writer when a document is available (unlocked)
Authorization	It refers to security policies regarding document handling, so that only authorized users can access to certain document contents
Logging	It is in charge of keeping track of activity on the documents after some abnormal situation involving security issues has occurred in the system. Typically, any unsuccessful authorization should dynamically activate the logging aspect to register user actions from that point on

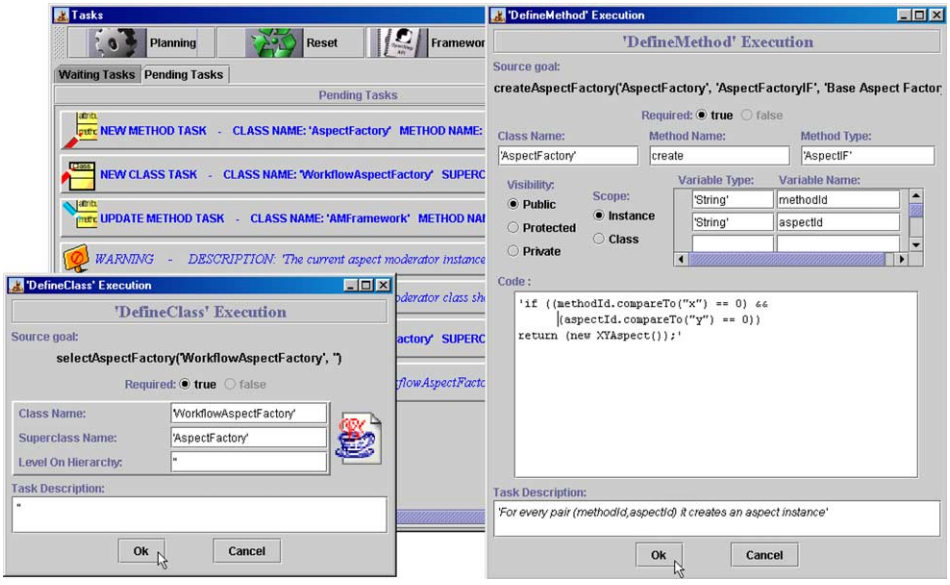

Fig. 3. A class-refinement task proposed by the Task Manager.

will enable the reuse of such a component on future developments, if the implemented functionality is required again. Nevertheless, the user can remove or reject any task without executing it.

At last, the application is shaped through several interactions of the user with the Task Manager. The execution of the tasks suggested by *Hint* generates a code skeleton of the main classes and methods, which the user can later fill in with more specific implementation details. Fig. 4 shows a class diagram for the workflow application, created using *Hint*.

4. Defining the instantiation knowledge

As we mentioned before, the core knowledge of *Hint* is based on the information provided by the designer through the *Smartbooks* documentation method (Ortigosa et al., 2000). *Smartbooks* considers the instantiation of an application from reusable assets as an activity based on a well-defined amount of basic instantiation tasks, for example: class specialization or method overwriting, among others. The method prescribes that the designer should describe the functionality provided by the reusable asset, how this functionality is implemented by different framework components, and provide rules to somehow constraint the way the software should be specialized. This special documentation constitutes what is called *instantiation schemes*.

Instantiation schemes can be graphically specified using a UML (Object Modeling Group, 2001) extension called Tasks and Object-Oriented Notation (TOON), to express design structures and instantiation activities associated with them. All these rules are directly associated with the concept of *tasks*, as programming activities to be carried out by the developer in order to use or specialize a given set of components to implement a specific function. The execution of these tasks will effectively end up with the code implementing the desired functionality, which in some cases can be automatically generated or, in the general case, will produce a template to be filled in by the programmer.

Instantiation schemes can be divided into two categories:

- *Specific schemes:* These schemes are written by the designer and describe instantiation knowledge particular of a specific reusable asset.
- *Generic schemes:* These schemes, on the other hand, describe knowledge common to the instantiation process, and they are used as building blocks to express specific schemes. Constraints about class specialization or component usage are typical examples of these schemes, but also, design patterns (Gamma et al., 1995) can be expressed as reusable schemes.

Fig. 5 shows a simple example of an instantiation scheme for the AMF, described using TOON. The white squares represent classes and the black ones represent instantiation tasks. The text on the upper left corner states the functionality described by the scheme. In the example, the scheme specifies that in order to have a functional proxy for a component, the proxy class should implement the *FunctionalProxyIF* interface and wrap the component. More precisely, the diagram prescribes that four tasks should be carried out by the user:

1. An *ImplementInterface* task that has to produce a subclass of *FunctionalProxyIF*.

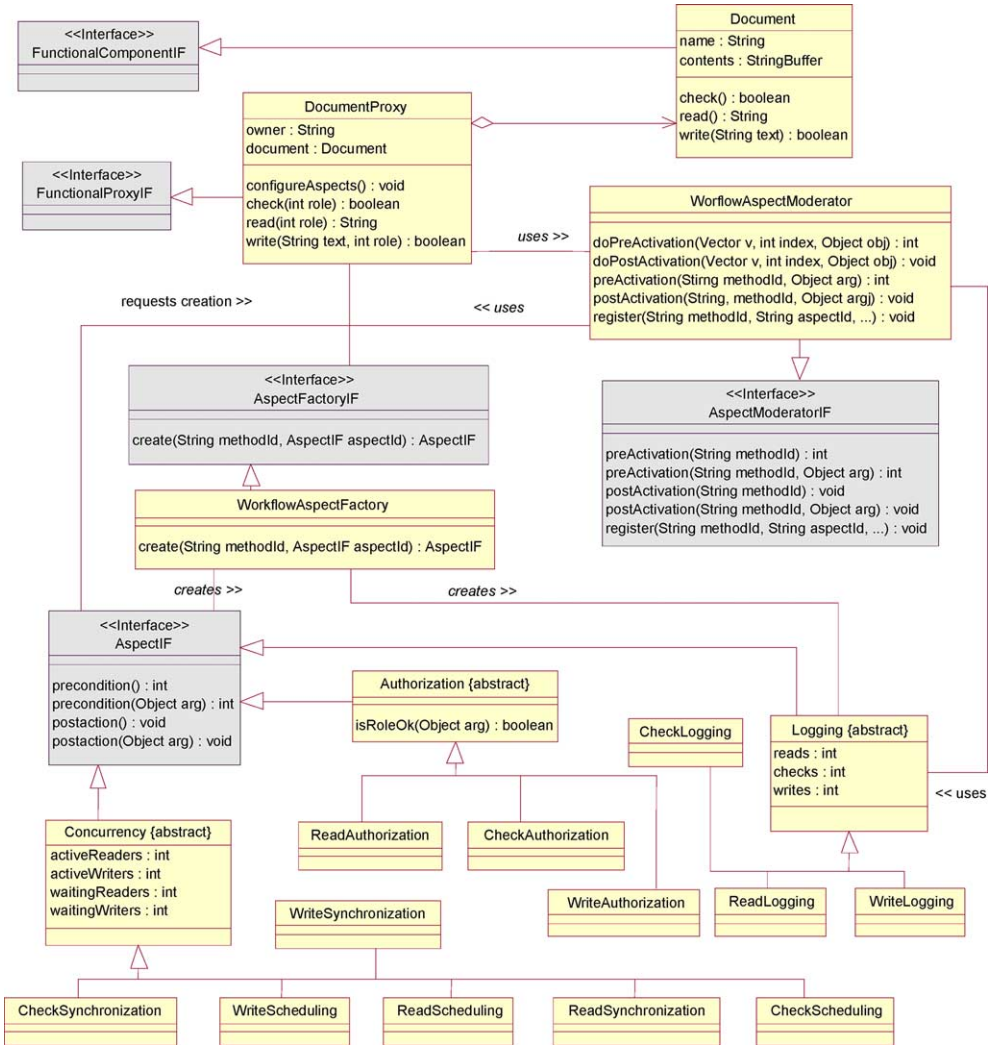

Fig. 4. The workflow application generated by *Hint* (gray boxes correspond to AMF classes).

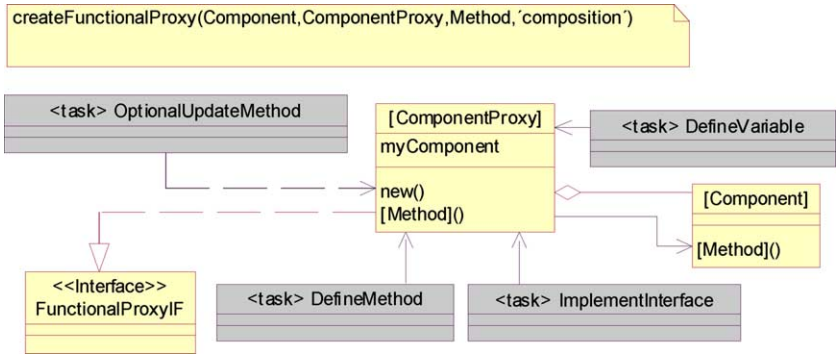

Fig. 5. Example of graphical instantiation scheme for the *Aspect-Moderator* framework.

2. A *DefineVariable* task, which must add a *myComponent* attribute in the proxy class.
3. A *DefineMethod* task in charge of overriding the method *Method( )* in the component (to incorporate additional behavior).
4. And finally, an *OptionalDefineMethod* task if any update to the constructor is required.

The kind of tasks presented in the example can be seen as basic tasks or *programming tasks*, because they

Table 2
List of some programming tasks supported by *Smartbooks*

Task	Type	Description
Define/UpdateClass	Pending	It adds/updates a new class definition (from an existing base class)
Define/UpdateMethod	Pending	It adds/updates a new method into a given class
Define/UpdateAttribute	Pending	It adds/updates a new attribute into a given class
ImplementInterface	Pending	It makes a class to implement a given interface
Warning	Pending	It shows a warning message explaining some framework constraint
AskSelection	Waiting	It presents several alternatives and asks the user for a choice
GetUserInput	Waiting	It asks the user to enter a string
SelectClass	Waiting	It asks the user to specify a class
SelectMethods	Waiting	It asks the user to specify a set of methods of a class

```
Scheme: createFunctionalProxy
Input: ComponentProxy, Component, Method
Output: []
Preconditions: definedClass(ComponentProxy), definedClass(Component),
               proxyDefinitionOption('Composition')
Postconditions: definedMethod(ComponentProxy, Method),
                wrapped(ComponentProxy, Component)
Body:
  do implementInterface(ComponentProxy, 'FunctionalProxyIF')
  do defineVariable(ComponentProxy, 'myComponent')
  do defineMethod(ComponentProxy, Method, 'This method should call Method in
     Component')
  do optionalUpdateMethod(ComponentProxy, 'new')
```

Fig. 6. Example of textual instantiation scheme for the *Aspect-Moderator* framework.

refer to programming activities associated with framework code. These tasks are classified as pending or waiting tasks, according to their role in the planning process. Table 2 shows a sample of other programming tasks supported by the environment.

Internally, an *instantiation scheme* is represented as a rule in the form *precondition-effect*s. The rule defines which preconditions are needed for the effects to be true. In every step of the planning algorithm, the planner tries to make true the preconditions of the rule whose effects (or at least one of them) are goals. It is important to note that the body of the rule is only evaluated when all their preconditions are true. The scheme of Fig. 6 shows the internal representation of the instantiation scheme presented in Fig. 5.

It must be noticed here, that the designer arbitrarily fixes the terms used to express functionality provided by the target framework. In this way, the tool can show to the user different kinds of functionality implemented by the framework so he can specify and also customize what is needed in his application.

5. The planner

This module provides some of the most important functionality of the *Hint* environment. From a list of functional requirements, it elaborates a list of required instantiation tasks based on the instantiation knowledge provided by the framework designer, particularly from the instantiation schemes. The central component of the Planner module is the *PHint* planning algorithm. This algorithm was developed on the basis of the UCPOP planning algorithm (Weld, 1994, 1998), and it was specifically adapted to fulfill the requirements of the framework instantiation domain.

One of the main requirements for the planning algorithm was to avoid making decisions before they were really needed. For example, if two tasks can be executed in any order, the algorithm should not impose any arbitrary sorting, but it should allow the user to choose which one executes first, or even executes them in parallel. This technique of delaying decisions as much as possible is known as *least commitment planning*.

It may be possible that, for a given set of functional requirements, the planner cannot find a suitable instantiation plan. Two reasons can produce an unsuccessful planning: either the functionality cannot be implemented using the framework or the documentation available is not enough to determine how this functionality can be implemented. In both cases, the planner should generate a partial plan for those requirements for which enough information is available. Once this partial plan has been generated, the user

can choose to begin executing the plan, or alter the initial requirements so that a complete plan can be produced instead. Even in the former case, he can also decide to return to the starting point and change the requirements after executing some tasks of the plan. In this situation, the planner should be able to build a new plan for the modified requirements, taking into account all the information provided by the user in the previous plan. If the user executed some tasks of the old plan, and those tasks are also part of the new plan, they must be reused. That is, the user should not be asked to execute twice the same task or to answer the same questions again.

The *PHint* algorithm works based on the scheme representation presented in Section 4. The algorithm is, basically, a loop that tries combinations of goals. If a plan cannot be built for the complete set of goals, the algorithm takes instead a subset of these goals. *PHint* works using backtracking, and eventually it will check every combination of goals until producing a plan for a given subset or returning an empty plan. A plan is built for a proper subset of the original goals, in the case that the framework documentation is not enough to completely describe how to implement the total required functionality. In other words, a complete plan for every goal is not possible, but some user tasks for implementing some goals are generated anyway.

6. Results and lessons learned

In order to evaluate the feasibility of the proposed approach, we have applied the *Hint* tool to assist the development of applications in two domains, namely: graphical interfaces using the *HotDraw* framework (Ortigosa and Campo, 2000) and aspect-oriented applications using the AMF framework (Diaz Pace et al., 2002). The preliminary results obtained from the comparison of a number of (assisted) designs against the solutions reported in the literature have revealed a very reasonable matching between them. For example, in the case of the workflow application presented in this paper, we contrasted the design given in Fig. 4 against the one described in (Constantinides et al., 2001). Anyway, these results should be taken as a partial insight of the potential of the approach, and more validation activities still remain to be done.

As regards the effectiveness of the approach, this is closely connected to the issue of framework documentation. It seems that a disadvantage of *Smartbooks* is that the tool cannot always derive the right instantiation actions if there is not enough knowledge available at the documentation repository. This aspect comes inherently associated with the approach, because it is assumed that the documentation provided suffice to get a correct framework assistance. However, the approach does not care neither about the relevancy/consistency of the instantiation schemes nor how they are acquired by framework developers. This may vary according to the facilities of the target framework and the developer's experience. On the other hand, the flexibility of instantiation schemes for documentation purposes enables further refinement of this representation through some domain language providing a textual vehicle to capture the functionality desired for a specific application. For example, in the case of the AMF, the mapping of schemes to rules let developers decide which is the best strategy to implement aspects on top of the framework. In addition, as the AMF is mostly composed of abstract classes and interfaces, it can be though more as programming model than a component-based framework. This feature makes the *Smartbooks* method particularly useful, because we can express many model constraints, not completely expressed through code structures, by means of instantiation schemes.

Besides, the current state of *Hint* environment presents a number of limitations. One of them is the way required functionality for a given application is specified in the Functionality Collector. Here, the use of textual menus to select functionality items may force the user to accommodate his needs to a vocabulary predetermined by the framework developer. As we have observed that, in practical cases, the developer usually defines these items on the basis of framework design models rather than on functionality-oriented framework services, a text-based strategy may diminish the capability of the approach to properly capture the user's needs. An alternative strategy to define functionality requirements is to complement menus with more expressive graphical models, for instance, UML diagrams.

As another drawback, the implementation of similar applications cannot be detected, loosing some opportunities to make the instantiation process simpler. For these reasons, we have started to explore the possibilities of enhancing the agent's reasoning capabilities with more advanced techniques such as case-based reasoning (Kolodner, 1993) and bayesian networks (Heckerman and Wellman, 1995).

7. Related work

The problem of framework usability has led to different documentation proposals. Some of these techniques include recipes and cookbooks, interface contracts, exemplars, or UML/F, among others. The first techniques designed for documenting frameworks were *recipes* and *cookbooks* (Pree, 1995). 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. An example of this type of documentation is (Krasner and Pope, 1988).

Currently, one of the most promising techniques is *active cookbooks* (Schappert et al., 1995), 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 him 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.

As the size and complexity of frameworks increased, more formal techniques were needed to represent framework structures. An example of formal documentation is *interface contracts* (Meyer, 1992), which are specifications of obligations, each one providing specification of a class interface and an 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. Moreover, the complexity of the resulting specifications makes them more adequate for automated interpretation.

Another approach is the one of (Gangopadhyay and Mitra, 1995) based on exemplars. 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. The user creates a new application by gradually adapting the exemplars according to the application requirements, being able to visualize in each step the result of these 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.

The work on UML-F (Fontoura et al., 2000) describes an approach to explicitly model framework variation points in UML diagrams by expressing the allowed structure and behavior of framework variation points with a specific language. The authors introduce a number of extensions to the standard UML to form a new profile called UML-F, especially useful for assisting framework development and instantiation. Variation points are modeled in terms of tagged values, applicable to both methods and classes. OCL specifications (Object Modeling Group, 2001) are used to describe pattern behavior that should be followed by the variation point instances. Besides, a tool assisting developers is given, based on the use of several design patterns, meta-programming and aspect-oriented programming to support various implementation models. Additionally, the UML-F semantics are formally described to allow the verification of the design, implementation and adaptation activities.

Analyzing the aforementioned approaches, it can be observed that techniques like exemplars and cookbooks (including active cookbooks) are good at providing procedural assistance, but the lack of design rationale makes it hard to adapt the documented framework in unanticipated ways. On the other hand, more passive approaches, especially those combining more than one documentation technique, seem to provide more flexibility. This usually comes at the price of imposing a greater overload on the framework user. This additional effort is more evident with novice users, who may need to spend a considerable time reading deeply into the documentation before being able to build applications. As regards the user experience, it should be considered that users with different knowledge level have different framework needs. None of the analyzed approaches provides the flexibility needed to support these types of conflicting requirements.

In the light of this discussion, the best approach seems to be the use of models combining active documentation with detailed design explanations, both formally and informally specified. Although this schema has been already proposed by some approaches built around hypermedia systems (Lajoie and Keller, 1994), they do not explain how this combination should be achieved beyond the use of hyperlinks to relate the different documentation parts. From a different perspective, the UML-F approach tries to support framework development and instantiation in a manner very similar to the one proposed by *Smartbook*s. The extensions to UML given by Fontoura et al. provide alternative mechanisms to the instantiation schemes. It is also important to note how the approach takes into account the selection of different implementation alternatives for a given framework design.

8. Conclusions

In this work, an agent-based approach aiming to provide better means to cope with application composition was presented. The use of this approach requires the documentation be produced according to the *Smartbooks* method. The *Hint* tool is able to derive an instantiation plan for building a given application on

top of a target technology. The approach can play an important role in helping different kinds of users to better understand what they really need of a framework when building their applications.

An additional contribution of this work is the demonstration that planning techniques can be useful to generate a sequence of programming tasks to guide a user to implement an application. These techniques are the core component of an agent, which is able to guide the instantiation process based on the functionality required for the application being implemented. Moreover, the assistance provided by the agent could be extended beyond the limits of what was anticipated in the design and documentation. Along this line, we have started to investigate the use of the *Smartbooks* method to provide guidance in the development of aspect-oriented applications, extending programming tasks to support higher-level design activities and mapping to specific implementation technologies.

References

Bosch, J., 2000. Design and Use of Software Architecture-Adopting and Evolving a Product-line Approach. Addison-Wesley.

Bradshaw, J., 1997. An introduction to software agents. In: Bradshaw, J.M. (Ed.), Software Agents. AAAI Press/The MIT Press, pp. 3–46 (Chapter 1).

Campo, M., Diaz Pace, A., Zito, M., 2002. Developing object-oriented enterprise quality frameworks using. Software Practice and Experience 32 (8), 837–843.

Constantinides, C., Bader, A., Elrad, T., Netinant, T., Fayad, M., 2000. Designing an aspect-oriented framework in an object-oriented environment. ACM Computing Surveys 32 (1), 41–41.

Constantinides, C., Skotiniotis, T., Elrad, T., 2001. Providing dynamic adaptability in an aspect-oriented framework. ECOOP 2001.

Demeyer, S., D'Hont, K., Steyaert, P., 2000. Consistent framework documentation with computed links and framework contracts. Computing Surveys (March).

Diaz Pace, A., Campo, M., Trilnik, F., 2002. Assisting the development of aspect-based multi-agent systems using the smartweaver approach. In: Proceedings SELMAS 2002, LNCS-Springer Special Volume on Software Engineering for Large-Scale Multi-Agent Systems.

Fayad, M., Schmidt, D., Johnson, R., 2000. Building Application Frameworks, Object-Oriented Foundations of Framework Design. Wiley Computing Publishing.

Fontoura, M., Pree, W., Rumpe, B., 2000. UML-F: a modeling language to object-oriented frameworks. In: Proceedings ECOOP 2000, vol. 1850 of LNCS. Springer, pp. 63–82.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley, Massachusetts.

Gangopadhyay, D., Mitra, S., 1995. Understanding frameworks by exploration of exemplars. In: Muller, H.A., Norman, R.J. (Eds.), Proceedings: 7th International Workshop on Computer-Aided Software Engineering. IEEE Computer Society Press, pp. 90–99.

Heckerman, D., Wellman, P., 1995. Bayesian networks. Communications of the ACM 38 (3), 27–30.

Helm, R., Holland, I., Gangopadhyay, D., 1990. Contracts: specifying behavioral compositions in object-oriented systems. In Proceedings of the OOPSLA/ECOOP '90 Conference on Object-oriented Programming Systems, Languages and Applications, October 1990, pp. 169–180. Published as ACM SIGPLAN Notices, vol. 25, number 10.

Johnson, R., 1992. Documenting frameworks using patterns. In Proceedings of the OOPSLA '92 Conference on Object-oriented Programming Systems, Languages and Applications, October 1992, pp. 63–76. Published as ACM SIGPLAN Notices, vol. 27, number 10.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J., 1997. Aspect-oriented programming. In: Mehmet, A.K., Satoshi, M. (Eds.), ECOOP'97—Object-Oriented Programming 11th European Conference, Jyväskylä, Finland. In: Lecture Notes in Computer Science, vol. 1241. Springer-Verlag, New York, NY, pp. 220–242.

Kolodner, J., 1993. Case-based reasoning. Technical Report, 1993. Also in Morgan Kaufmann Publishers.

Krasner, G., Pope, S., 1988. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming 1 (3), 26–29.

Lajoie, R., Keller, R., 1994. Design and reuse in object-oriented frameworks: patterns, contracts and motifs in concert. In: Proceedings 62nd Congress of the ACFAS, Canada.

Meyer, B., 1992. Applying "design by contract". Computer 25 (10), 40–51.

Object Modeling Group, 2001. Unified Modeling Language Specification, Version 1.4.

Ortigosa, A., Campo, M., 2000. Using incremental planning to foster application framework reuse. International Journal on Software Engineering and Knowledge Engineering 10 (4), 433–448, World Scientific Publishing Company.

Ortigosa, A., Campo, M., Moriyón, R., 2000. Towards agent-oriented assistance for framework instantiation. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA-00), October 15–19, 2000 ACM Sigplan Notices, vol. 35.10. ACM Press, NY, pp. 253–263.

Pree, W., 1995. Design Patterns for Object-Oriented Software Development. Addison-Wesley. ACM Press Books. ISBN 0-201-42294-8.

Schappert, A., Sommerland, P., Pree, W., 1995. Automated support for software development with frameworks. In: Proceedings SSR'95 ACM SIGSOFT Symposium on Software Reusability.

Weld, D., 1994. An introduction to least commitment planning. AI Magazine 15 (4), 27–61.

Weld, D., 1998. Recent advances in AI planning. Technical Report TR-98-10-01. Department of Computer Science and Engineering, University of Washington, October 1998.

Marcelo R. Campo is full professor in the Computer Science Department and head of the ISISTAN Research Institute at the UNICEN University (Tandil, Buenos Aires, Argentina). He is also member of CONICET-Argentina.

J. Andrés Díaz Pace is a research assistant in the Computer Science Department and the ISISTAN Research Institute at the UNICEN University (Tandil, Buenos Aires, Argentina). He is also a doctoral candidate with the Faculty of Sciences at the UNICEN University.

Federico U. Trilnik is a research assistant in the Computer Science Department and the ISISTAN Research Institute at the UNICEN University (Tandil, Buenos Aires, Argentina). He is also a doctoral candidate with the Faculty of Sciences at the UNICEN University.