Hint: A planning-based tool for framework instantiation

Federico U. Trilnik and J. Andrés Díaz Pace

ISISTAN Research Institute, Facultad de Cs. Exactas, UNICEN University
Campus Universitario, (B7001BOO) Tandil, Buenos Aires, Argentina
Also CONICET - Argentina

E-mail: { ftrilnik, adiaz }@exa.unicen.edu.ar

Abstract. Object-oriented frameworks are, perhaps, the most important reuse technology today available. This is so, because they enable the reuse of both code and design. However, the effective use of frameworks to instantiate new applications represents one of the most limiting factors of this reuse technology. Therefore, tools to support the process of framework instantiation are increasingly necessary. In this line, we present *Hint*, an environment for assisting the instantiation of Java frameworks based on software agent technology. Actually, *Hint* is the evolution of a previous work in the subject, and it was built around a software agent that starts asking the user to select the functionality needed for a new application and after that elaborates a sequence of programming activities that should be carried out in order to implement it. In order to enable the agent to do its work, the framework must be documented following the *Smartbooks* documentation method. The most relevant contribution of this work is the use of planning techniques to guide the execution of instantiation activities according to the framework design.

Keywords: object-oriented frameworks, framework documentation, software agents, planning.

1. Introduction

Framework usability is one of the most limiting factors of framework-based reuse technology. Frameworks [8] embody a generic design for a family of applications, and they can be seen as a tradeoff between a general and a flexible solution. A general solution can deal, without changes, with different variants of a given problem. On the other hand, a flexible solution is one that through small changes on its organization, can be adapted to solve those variants. Flexible design structures, that is, structures that enable the adaptation of a general behavior, represent the essence of a framework. However, these structures often imply more complex designs and, therefore, designs harder to understand. For this reason, much effort has been dedicated in the last ten years to the development of more powerful documentation techniques and tools, in order to facilitate the framework understanding process [2]. Nevertheless, none of the developed techniques [6][10] can be completely adapted to the different types of framework users, especially if considering the variations on knowledge and experience these users usually have [9]. 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 the framework 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 adapt the framework 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 frameworks can bring to enhance software development.

This limitation led us to look for more intelligent tools that could effectively help users in the instantiation of a given framework. In this context, an ideal supporting tool should allow a framework user to describe the required functionality for a specific application and then the tool should automatically generate an application implementing it. Unfortunately, it is not always possible to provide that kind of tools, mainly because in order to instantiate an application, some programming activities are usually needed, depending on the application domain or framework

maturity. For this reason, an intermediate approach is to think about tools having the necessary knowledge to show the user what should be done to implement a given application using the framework. In this sense, intelligent agent technology represents an interesting alternative to explore. *Intelligent Software Agents* [1] is a research area were important advances have been achieved in the last years [5], providing new ways for analyzing, designing and implementing software systems. Loosely defined, an intelligent agent is a software component that embodies knowledge about a given subject or domain and uses this knowledge in an autonomous way to satisfy its design goals.

In this work we introduce *Hint*, a *Java* tool designed to provide semi-automated support to the process of framework instantiation based on an assistance intelligent agent [14]. This agent uses specific knowledge to assist the process of framework instantiation through least commitment planning techniques [15]. The agent provides a developer with the list of functions that can be implemented using the framework. Once the user has selected the functionality he is interested in, the agent uses the instantiation knowledge to derive an instantiation plan. The plan consists of a sequence of instantiation tasks (e.g., subclassing, method implementation, an so on) that the user should carry out in order to have implemented his desired application. The knowledge needed for implementing this functionality must be provided by the designer using the *Smartbooks* method for documenting frameworks. *Smartbooks* extends traditional techniques for framework documentation with rules that describe the way a given functionality can be obtained using the framework. These rules can be expressed using an *UML*-like visual notation, giving the opportunity of being integrated with other existent documentation tools. To carry out the process of plan creation we have developed a specialized planning algorithm, called *PHint*. In this paper we present the main characteristics of the tool, with special focus on developed planning algorithm.

The rest of the paper is organized as follows. Section 2 covers the *Smartbooks* documentation method. Section 3 presents the architecture of the *Hint* environment and describes *PHint*, the planning algorithm used by *Hint*. Finally, section 4 rounds up the conclusions of the work.

2. *SmartBook*s: Defining the Instantiation Knowledge

As we mentioned before, the functionality of *Hint* is based on the information provided by the framework designer through the *Smartbooks* documentation method [14]. *Smartbooks* considers the instantiation of frameworks 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 framework designer should describe the functionality provided by the framework, how this functionality is implemented by different framework components, and provide rules to somehow constraint the way the framework is specialized. This special documentation constitutes what is called *instantiation rule*s.

By means of instantiation rules, the *Hint* agent is able to provide 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 his application is supposed to do. Relying on the information about application functionality, the agent can elaborate a plan for framework instantiation, and then the user has just to execute (perhaps including some modifications) the list of tasks that composes the plan.

Instantiation rules can be graphically specified using a UML [13] extension called *TOON* (Tasks and Object-Oriented Notation), to express framework 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 correctly instantiate a target framework. The execution of these tasks will effectively generate code implementing the desired functionality.

2.1 Instantiation Rules

This kind of rules documents different ways to implement predetermined functional requirements and in the same time provides support to assist the user in this process. Instantiation rules can be divided into two categories:

- *Framework-specific rules:* These rules are written by the framework designer and describe instantiation knowledge particular of each framework
- *Generic rules:* These rules, on the other hand, describe knowledge common to the instantiation process of many frameworks, and they are used as building blocks to express framework-specific rules.

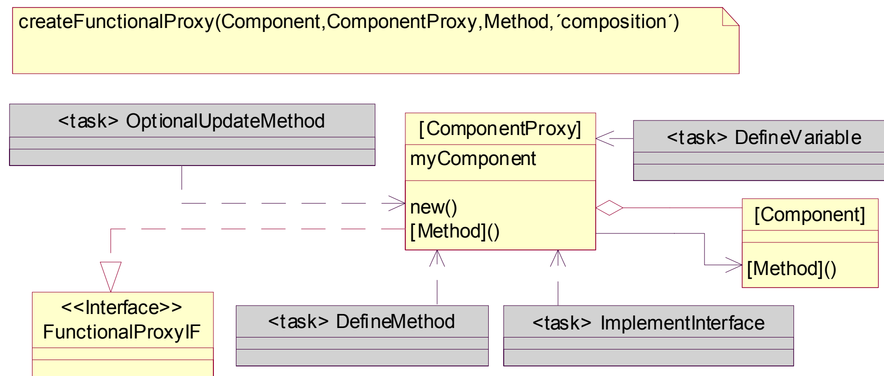

Figure 1. Example of an instantiation rule for the *Aspect-Moderator* framework using *TOON*

Figure 1 shows a simple example of an instantiation rule for the *Aspect-Moderator* framework [4], described using *TOON*. This example is explained in more detail in [3]. 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 rule. In the example, the rule 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*.
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 refer to programming activities associated with framework code. These tasks are classified as pending or waiting tasks, according to their role on the planning process. Table 1 shows a sample of other programming tasks supported by the environment.

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

Task	Type	Description
DefineClass	Pending	It adds a new class definition (from an existing base class)
DefineMethod	Pending	It adds a new method into a given class
DefineAttribute	Pending	It adds a new attribute into a given class
UpdateClass	Pending	It modifies the definition of a class
UpdateMethod	Pending	It modifies the definition of a method
UpdateAttribute	Pending	It modifies the definition of an attribute
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
SelectMethod	Waiting	It asks the user to specify a method of a class
SelectMethods	Waiting	It asks the user to specify a set of methods of a class

Internally, an *instantiation rule* is represented in the form *precondition-effect*s. A rule states 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. Figure 2 shows the representation of the instantiation rule presented in the example (see figure 1).

```
Task: 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´)
```

Figure 2. Instantiation rule representation.

3. The *Hint* architecture

Hint is a special tool in charge of collecting all the *Smartbooks* documentation and assisting the user through the process of framework instantiation. The tool architecture comprises several components with different responsibilities. For the sake of simplicity, we just give an outline of them. A detailed description of these issues is given in [14]. The components are the following:

- A documentation tool used by framework developers to write the framework documentation.
- A rule generator based on the framework description, which generates a Prolog representation of the rules.
- A functionality collector (wizard) to help the framework user describe the functionality required for the application currently being developed.
- A planning module using least commitment planning techniques to build instantiation plans.
- A task manager keeping track of the tasks involved in instantiation plans.

During the planning phase, a specially designed *UCPOP*-like algorithm is used. In the next sub-sections both the planning algorithm and the task manager component are described in more detail.

3.1 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 functional rules. The central component of the *Planner* module is the *PHINT* planning algorithm. This algorithm was developed to support the *Smartbooks* method on the basis of a well-known planning algorithm, 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*. One of the most popular algorithms in this category is the *UCPOP* algorithm [15], which was quite well suited to be used in the framework instantiation domain. Nevertheless, there are some special properties of the domain that made necessary the design of the *PHINT* algorithm:

- If there is not enough information to generate a plan to completely implement the required functionality, the planner must generate a plan that can guide the framework instantiation according to the information available
- Planning inputs can be changed or new information can be added, and the plan should be updated accordingly.

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 which enough information available. Once this partial plan has been generated, the user can choose to change the initial requirements so that a complete plan can be produced instead, or to begin executing the plan. Even in the later case, he can 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. Because of these properties, *PHINT* is said to produce partial and incremental planning.

One additional difference with traditional planning algorithms, like for example *UCPOP*, is that *PHINT* allows the final plan to contain unbound variables, i.e. variables without an associated value. This results useful in situations where it is required to leave some decisions to the framework user, like for example the class name attribute of a task executed in a given step of the instantiation process.

3.1.1 PHINT

The *PHINT* algorithm works based on the rule representation presented in Section 2. 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.

PHINT takes three arguments as inputs:
1. A plan (initially null)
2. A set of goals to be satisfied
3. A set of rules describing actions that can be used to satisfy the goals

These rules correspond with traditional planning operators, which are additionally provided with an executable body where the tasks associated with the operator are hooked. In this way, any operator comprises a set of preconditions for executing the action, a set of postconditions that hold when the action is effectively executed, plus a body of *Smartbooks* tasks. In figure 3 we can see a partial graphical representation of the planning execution.

The plan itself is composed by six elements (they all are empty in the initial plan):
1. A list of instantiated actions
2. A list of partial orderings
3. A set of causal links
4. A list of variables with their corresponding values (bindings)
5. A list of pending tasks created through the planning process
6. A list of goals that are not considered in the plan

Algorithm 1 shows the basic steps of *PHINT*. The first step of *solvePlanning* tests if the agenda is empty, and if so, it returns the current plan. If the agenda has more items, the algorithm selects a goal from it and tries to satisfy it. There are several possibilities for the selected goal, as it can see from the case options. If the goal corresponds either to a single operator (a predicate) or a special equality operator, it will be processed later by *solvePredicate* or *handleOperator* respectively. In *solvePredicate* we deal with the selection of actions matching these goals (preconditions). The next steps handle multiple goals. In the case of a conjunction of goals, the algorithm tries to get an acceptable plan for every goal of the conjunction by calling *solvePlanning* recursively. If any of these goals fails, the algorithm undoes its actions accordingly. In the case of a disjunction of goals, the algorithm is just interested in one of the goals to be true. As soon as this happens, the whole expression is true and the planning proceeds with other goals. When all the preconditions of a given action have been considered, its corresponding tasks are evaluated.

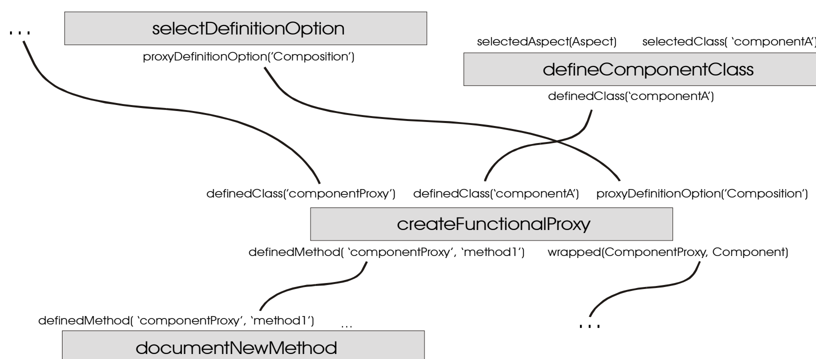

Figure 3. Planning execution example

```
class PHint {
…
   boolean solvePlanning(<actions, order, causalLinks, bindingTable>, agenda ) {
      if (agenda.hasNextItem()) {
         currentGoal = agenda.removeFirstItem();
         switch (currentGoal) {
          case OPERATOR:
               ok = handleOperator(currentGoal,agenda);
               break;
          case SIMPLE_GOAL:
               ok = solvePredicate(<…>,currentGoal,agenda);
               break;
          case CONJUNTION:
               agenda.addFirst(currentGoal.conjuntionItems());
               ok = solvePlanning(<…>,agenda);
               if (!ok) agd.remove(currentGoal.conjuntionItems());
               break;
          case DISJUNTION:
               agenda.addFirstItem(currentGoal.firstItem());
               satisfied = solvePlanning(<…>,agenda);
               if (!ok) {
                  agenda.removeFirstItem();
                  agenda.addFirstItem(currentGoal.secondItem());
                  ok = solvePlanning(<…>,agenda);
                  if (!ok)     agenda = agenda.removeFirstItem();
               }
               break;
          case FINISHED_PRECONDITIONS:
               // When preconditions are done, it proceeds to execute tasks
               ok = Planning.executeTasks(currentGoal.getAction(), bindingTable);
               if (ok) {
                  ok = solvePlanning(<…>,agenda);
                  if (!ok) {
                       tasks = currentGoal.getAction().getExecutableTasks();
                       Planning.undoExecutedTasks(tasks, bindingTable);
                  }
               }
         } // End switch
         if (!ok)    agd.addFirstItem(currentGoal);
         return (ok);
      } // End first if
      else return (true);
   } // End solvePlanning
…
} // End class PHint
```

Algorithm (part 1)

Algorithm part 2 describes both *handleOperator* and *solvePredicat*e. In *handleOperator*, the algorithm handles codesignated and non-codesignated relationships among variables, and then updates the binding table accordingly. Finally, *solvePlanning* is called again to continue with the planning. Things are a little more complicated with *solvePredicat*e. Here, *PHINT* looks for an action that satisfies the current goal. With this purpose, it takes the set of actions whose postconditions include the current goal, and then evaluates the actions until one of those fits well into the current plan. This part corresponds with the *choose* function of classical planning. Note that the algorithm looks firstly for an action being previously instantiated, that is it tries to find if the work was already done by some action of the plan. If one of these previous actions can be used, then it is selected. In the case none of the instantiated actions is useful, the algorithm looks for a new action describing how to satisfy the current goal. It takes the first action available, but if it fails to satisfy a goal in the future, the algorithm will backtrack to this point and a different action (if any) will be taken. At last, we need to be sure that the selected action meets the ordering requirements of the plan, by executing *solveWithActio*n. If the action is an old action, it just requires the elimination of the current goal from agenda. It also updates the list of bindings and the list of

actions. On the other hand, if it is a new action, its preconditions are added to agenda. Again, bindings and actions are updated. The algorithm checks the action for ordering and protection of causal links, so that the necessary relationships are added to the partial ordering list. Finally, the algorithm makes a recursive invocation of *solvePlanning*.

```
class PHint {
…
   boolean solvePredicate(<oldActions, ...>, currentGoal, agenda) {
      // Actions already included in the current plan
      posibleActions = selectOldActions(currentGoal);
      // All the actions of the domain
      posibleActions = posibleActions U selectNewActions(currentGoal);
      solved = false;
      while ((!solved)&&(posibleActions.hasNext())) {
         action = posibleActions.next();
         solved = solveWithAction(<>,agenda,currentGoal,action);
      }
      return (solved);
   }

   private boolean solveWithAction(<oldActions, order, causalLinks, bindingTable>, agenda,
currentGoal, action) {
      // It verifies that a given action fits well into the current plan
      actionEffect = action.getEffects(currentGoal);
      newBindingTable = bindingTable;
      if (newBindingTable.propositionBinding(actionEffect,currentGoal) {
         // The order is updated
         newOrder = order U Order(action, currentGoal.getAction());
         // Links are checked for threats and possibly repaired
         // according with the added action
         repaired = repairLinks(threatenedLinks(action),action,newOrder);
         while (repaired && (oldActions.hasNext())) {
            // It also checks threats originated by existing actions
            oldAction = oldActions.next();
            repaired = repairLinks(threatenedLinks(oldAction),oldAction,newOrder);
         }
         if (repaired) {
            newCausalLinks = causalLinks U Link(action,currentGoal,currentGoal.getAction());
            newActions = addAction(actions, action);
            // It updates the goals of the agenda
            newAgenda = addGoals(agenda, action);
            solved = solvePlanning(<newActions, newOrder, newCausalLinks, newBindingTable>,
newAgenda);
         } else solved = false;
         return solved;
      } else return false;
   } // End solveWithAction

   private boolean handleOperator(operatorPredicate, agenda) {
      arg1 = operatorPredicate.firstArg();
      arg2 = operatorPredicate.secondArg();
      switch (operatorPredicate) {
         case NOT_EQUAL:
               ok = bindingTable.noCodesignate(arg1, arg2);
               if (ok) ok = solvePlanning(agenda);
               break;
         case EQUAL:
               ok = bindingTable.codesignate(arg1, arg2);
               if (ok) ok = solvePlanning(agenda);
               break;
      }
      return ok;
   } // End handleOperator
...
} // End class PHint
```

Algorithm (part 2)

```
class Planning {
   Set pendingTaskSet, waitingTasksSet;

  // Method invoked to execute tasks (used by PHint)
  boolean executeTasks(action, bindingTable) {
     executableTasks = action.getExecutableTasks();
     while (executionResult & executableTasks.hasNext()) {
        task = executableTasks.next();
        switch (task.getType()) {
           case DATABASE_QUERY:  // SUBCLASS_OF, CHILD_OF, DONE, EXECUTED, EXIST_CLASS ...
               result = taskOperatorsManager.executeQuery(action,pred,executableTasks);
               break;
           case WAITING_TASK:
               executionResult = this.executeWaiting((WaitingTask)task);
               break;
           case PENDING_TASK:
               executionResult = this.storePending(task);
               break;
        }
        if (result) executedTasksStack.push(task);
     }
     if (!result) undoExecutedTasks(executedTasksStack.inverse(), bindingTable);
     return executionResult;
   } // End executeTasks

   boolean executeWaiting(waitingTask) {
     oldEquivWaitingTask = waitingTasksSet.getEquivalentTask(waitingTask);
     waitingTasksSet.addTask(waitingTask);
     if (oldEquivWaitingTask!=null) {
        waitingTask.setExecutionResult(oldEquivWaitingTask.getExecutionResult());
     } else {
        taskManager.executeWaitingTask(waitingTask);
        // It waits until the tasks is effectively executed by the user
        while (!waitingTask.isFinished());
     }
     // The result is bound with the returned variable
     result = waitingTask.bindResult(bindingTable);
     return result;
   } // End executeWaiting

   boolean storePending(pendingTask) {
     oldEquivPendingTask = pendingTasksSet.getEquivalentTask(pendingTask);
     // If an equivalent task doesn’t exists, it is stored in the set
     if (oldEquivWaitingTask==null)
        pendingTasksSet.addTask(pendingTask);
     return true;
   } // End storePending

   void undoExecutedTasks(executableTasks, bindingTable) {
    while (executableTasks.hasNext()) {
        task = executableTasks.next();
        switch (task.getType()) {
           case DATABASE_QUERY:
                taskOperatorsManager.undoExecutedQuery(task);
                break;
           case WAITING_TASK:
                task.undoBindings(bindingTable);
                break;
           case PENDING_TASK:
                pendingTasksSet.removeTask(task);
                break;
        }
    }
  } // End undoExecutedTasks

...
} // End class Planning
```

Algorithm (part 3)

Algorithm 3 keeps track of all the tasks generated as a result of executing *PHINT* actions. This part is mostly independent of the particular type of planning algorithm used by the tool. The first method, *executeTasks*, is responsible for executing those tasks that require an immediate response (queries to the repository and waiting tasks) and also storing those tasks that may be later executed by the user (pending tasks). In the case of queries, the algorithm simply accesses the classes repository and returns the result of the evaluated query. Likewise, waiting tasks are managed by *executeWaitin*g. The algorithm waits until the user executes the task and provides a concrete result. Then, planning is resumed and the control returns to *PHINT*. If the task is a pending task, it is just added to a list by *storePending* and passed later to the *TaksManage*r. Note that when both pending and waiting tasks are generated, the planning checks with the current task set that these new tasks are not actually duplicated (perhaps because some tasks were generated in previous stages). Nonetheless, if any task is not successfully executed, the algorithm undoes partial execution results (bindings and stored pending tasks) and goes back to *solvePlanning* in order to select another candidate action.

The *PHINT* planning algorithm is an enhanced version of the original planning algorithm presented in [14]. Planning is now implemented in Java, and it has been tuned in order to get a better performance. Moreover, *PHINT* is able to take advantage of *TOON* diagrams, and it incorporates this information to prune the number of action choices. The sequences of instantiation tasks defined in these diagrams are seen as partially instantiated action sequences that should be met by the final plan according to the framework prescriptions, thus avoiding an unnecessary overwork during the planning process. In this line, we are currently analyzing different alternatives to become planning more efficient and powerful.

3.2 Task Manager

During the instantiation process, the user interacts with the task manager to build applications. The task manager is responsible for coordinating tasks, so that they may be executed, interrupted in order to work on other tasks, or even cancelled at any time.

A task execution generally involves a form (associated with the task type) that the user has to fill in. For example, a new-class task execution opens a form where the user specifies parameters such as class name, super-class name and documentation notes. The user can also obtain information about a given task, for example the specific rule that originates the task or other parameters. Finally, a tasks rejection implies that the user had refused the proposed plan and, if possible, the planner should provide an alternative plan where this task is not included.

The tasks generated by the task manager can be classified in two categories, optional tasks and mandatory tasks. This distinction is used to difference between tasks that, according to the documentation, are not indispensable to obtain the specified functionality and essential tasks. Generally, optional tasks give information about programming activities to complement those tasks that must be executed.

In addition, the task manager checks that tasks are not duplicated upon creation, so that the user does not have to execute the same task twice. This is particularly important when instantiation plans are revaluated, otherwise the user would have to do the same work many times. As a final remark, we could mention that by means of the task manager interface, the user is able to change the rules and goals, initialize and execute the planner. In figure 4 shows a possible execution snapshot of the task manager (optional tasks appear in italic).

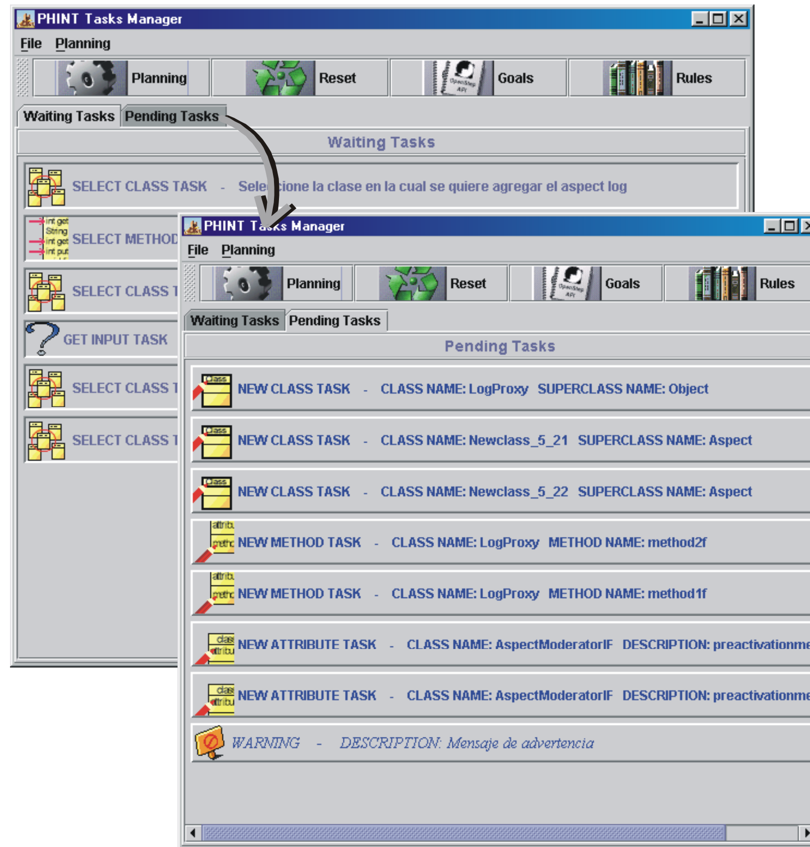

Figure 4. Task Manager during instantiation process execution.

4. CONCLUSIONS AND FUTURE WORK

The main contribution of this work is to show how the instantiation process can be represented as a sequence of programming 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.

An effective improvement on the guidance of framework-based application development can be achieved by centering the assistance on the application functionality. This is especially useful with novel users, have little knowledge about the framework design. Nevertheless, the approach is flexible enough to assist more expert users. If a user decides to ignore the suggested plan, the rules are still useful to guide him/her to implement the application consistently with the framework design.

We have also mentioned a visual formalism to specify the rules, as an extension of conventional UML diagrams. At this moment, a basic CASE environment integrated with the *Smartbooks* engine has been developed. We hope to improve these capabilities in the future, for example, adding more UML models. As regards the use of agents and planning techniques to generate sequences of programming tasks to guide a developer to implement applications on top of frameworks. Moreover, the agent's assistance agent can be extended to deal with more abstract design activities. In this line, we have elaborated an example where the *Smartbooks* method is applied to aspect-

oriented frameworks [3, 7, 11]. There, we have discussed how the notion of programming tasks may be expanded to higher-level activities supporting both design tasks and mapping to specific implementation technologies.

We have started to explore the possibilities of enhancing the agent's reasoning capabilities with more advanced learning techniques. For example, case-based reasoning (CBR) [12] could be helpful to utilize old instantiation plans, adapting them to particular details of the new instantiation problem.

Finally, one of the drawbacks of the proposed approach is the additional burden for the framework developer. Any documentation technique implies some effort from developers, but in this case, the more complete the instantiation guide provided the more details about the structure framework must be specified. However, considering that framework developing is a hard and time-consuming task, and its success is highly tied to framework usability, producing good framework documentation is an activity that worths investing this effort.

REFERENCES

1. Bradshaw, J. *Software Agents*. AAAI Press, Menlo Park, USA. 1997

2. Butler G. and Denommée P. *Documenting frameworks*. In Proceedings of the Eighth Workshop on Institutionalizing Software Reuse, 1997.

3. Campo M., Diaz Pace J. and Trilnik F. *Smartweaver: An Agent-based Tool for Aspect-Oriented Development*. In proceedings of SECMAS 2002 (International Workshop on Software Engineering for Large-Scale Multi-Agent Systems) in conjunction with ICSE 2002. Orlando, Florida (USA), May 2002.

4. Constantinides C., Bader A., Elrad T., Fayad M. *Designing an Aspect-Oriented Framework*. Computing Surveys 32(1es):41. 2000.

5. Demazeau, Y., Müller, J. (eds.). *Decentralized AI* – Proceedings of the First European Workshop on Modeling Autonomous Agents in a Multi-Agent World (MAAMAW'89). Elsevier Science B.V. Amsterdam, Netherlands. 1990

6. Demeyer S., De Hondt K., and Steyaert P. *Consistent framework documentation with computed links and framework contracts*. Computing Surveys, 1999. To appear in March 2000.

7. Díaz Pace A., Campo M. *Analyzing the Role of Aspects in Software Design*. Communications of the ACM, Vol. 44 No. 10. 66-74. October 2001.

8. Fayad M., Schmidt D., Johnson R. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Wiley Eds. 1999

9. Helm R., Holland I., and Gangopadhyay D. *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems.* In Proceedings of the OOPSLA/ECOOP '90 Conference on Object-oriented Programming Systems, Languages and Applications, pages 169–180, October 1990. Published as ACM SIGPLAN Notices, vol. 25, number 10.

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

11. Kickzales G., Lamping J., Mendhekar J., Maeda C., Videira Lopes C, Loingtier J., Irwin J. *Aspect-Oriented Programming*. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finlad. Springer-Verlag LNCS 1241. June 1997.

12. Kolodner J. *Case-based reasoning*. Technical report, 1993. Also in Morgan Kaufmann Publishers, 1993.

13. Object Modeling Group. *Unified Modelling Language Specification, version 1*.4, September 2001.

14. Ortigosa A., Campo M. *Towards Agent-Oriented Assistance for Framework Instantiation*. Proceedings of OOPSLA 2000, October 2000.

15. Weld D. *An Introduction to Least Commitment Planning*. AI Magazine, Summer/Fall 1994.