Using UML-FW to Enhance Framework Development: a Case Study in the Local Search Heuristics Domain

Marcus Fontoura♣, Carlos J. Lucena*, Alexandre Andreatta*,
Sérgio E. Carvalho*, and Celso C. Ribeiro*

♣ Department of Computer Science, Princeton University
35 Olden Street, Princeton, NJ, U.S.A., 08544
fontoura@cs.princeton.edu

* Computer Science Department, Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
Rua Marquês de São Vicente, 225, 22453-900 Rio de Janeiro, Brazil
{lucena, andreatt, sergio, celso}@inf.puc-rio.br

PUC-Rio.Inf.MCC 34/99 Dezembro, 1999

ABSTRACT

Currently, frameworks are most commonly represented through design diagrams written in standard object-oriented analysis and design methods (OOADMs). However, these design notations do not provide elements for identifying the frameworks variation points and how the instantiation should be performed. Therefore, users have to rely on extra documentation in order to create framework instances, which is not always available. This work proposes an extension to UML that explicitly represent all the information required for framework development and instantiation. The approach is illustrated through a large real-world framework for local search heuristics.

KEY WORDS: object-oriented frameworks, design notation, UML-FW, framework development, framework instantiation.

RESUMO

Atualmente, Frameworks são representados através de diagramas de design escritos em alguma notação orientada a objetos. Entretanto, estas notações não oferecem elementos para a identificação dos pontos de flexibilização nos Frameworks e nem como a instanciação deva ser realizada. Este trabalho propõe uma extensão para UML de forma a explicitar todas as informações necessárias para o desenvolvimento e manutenção de Frameworks. Esta abordagem é ilustrada através de um estudo de caso de um Framework para buscas heurísticas.

PALAVRAS CHAVES: object-oriented frameworks, notação de design, UML-FW, desenvolvimento de framework, instanciação de framework.

INTRODUCTION

A framework is a collection of several fully or partially implemented components with largely predefined cooperation patterns between them. A framework implements the software architecture for a family of applications with similar characteristics [22], which are derived by specialization through application-specific code. Hence, some of the framework components are designed to be replaceable. We call these components the variation points or hot-spots [23] of the framework. An application based on such a framework not only reuses the frameworks source code, but more important, its architecture design. This amounts to a standardization of the application structure and allows a significant reduction of the size and complexity of the source code that has to be written by developers who adapt a framework.

Recent standardization efforts of the Unified Modeling Language (UML) [21, 28] offer a chance to harness UML as notational basis for framework development projects. UML is a multi-purpose language with many notational constructs and a built-in extension mechanism. However, the current standard UML does not provide adequate constructs to model frameworks. There is no indication in UML design diagrams what are the variation points and what are their instantiation constraints. Fortunately, UML provides extension mechanisms that allow us to define appropriate labels and markings for the UML model elements. Based on these mechanisms we have defined a new UML profile [5] called UML-FW especially useful for framework description, both during development and instantiation of a framework.

This paper describes UML-FW and illustrates how it can be applied to a large framework in the local search heuristics domain, namely the Searcher framework [1], assisting the development and instantiation activities. This case study is a condensed version of a real-application of UML-FW. Several other projects have already been successfully developed with UML-FW, as reported in [7, 9].

The rest of this paper is organized as follows: Section 2 describes UML-FW and how it can be applied to model a simple example in the web-based education domain. Section 3 describes the local search heuristics domain and presents the Searcher framework using the OMT [27] design language (as it was first described in [1]). Section 4 revisits the framework description using UML-FW, and discusses the benefits of this representation. Section 5 describes some related work. Finally, section 6 presents our conclusions and future research directions.

1. UML-FW: MODELING FRAMEWORKS AT DEISGN LEVEL

Standard OO design methods, like OMT [27] and UML [21, 28], provide static (class and object diagrams) and dynamic (activity diagrams, interaction diagrams, state-chart diagrams, and state diagrams) views of a system design. Figure 1 is a representation of a student sub-system of a web-based education framework [8] in UML, where (a) shows a static view of the system (class diagram) and (b) shows a dynamic view (interaction diagram). In the dynamic view we show the interaction between an instance of each of the two classes.

The *showCourse()* method is the one responsible for controlling the application flow: it calls *selectCourse()*, which allows the student to select the desired course, *tipOfTheDay()*, which shows a start-up tip, and finally *showContent()* to present the content of the selected course.

The method *selectCourse()* is the one responsible for selecting the course the student wants to attend. It is a framework variation point since it can have different implementations in different web-based applications created within the framework (framework instances). Different examples of common selection courses mechanisms include: to require a student login or not, to show the entire list of available courses or just the ones related to the student major, and to show a course preview or not, and so on. There are an infinite number of possibilities that depend on the framework use.

Figure 1 shows *selectCourse()* as an abstract method of an abstract class, *SelectCourse*. The framework user has to create a subclass from *SelectCourse* and then provide a concrete implementation for *selectCourse()*. The problem with this representation is that there is no indication that this method is a variation point in the design diagrams. There is not also any indication of how the variation point should be instantiated. Although the name of the abstract method *selectCourse()* is italicized this notation is not an indication of a variation point, rather it indicates an abstract method; an abstract method does not necessarily have to be a variation point.

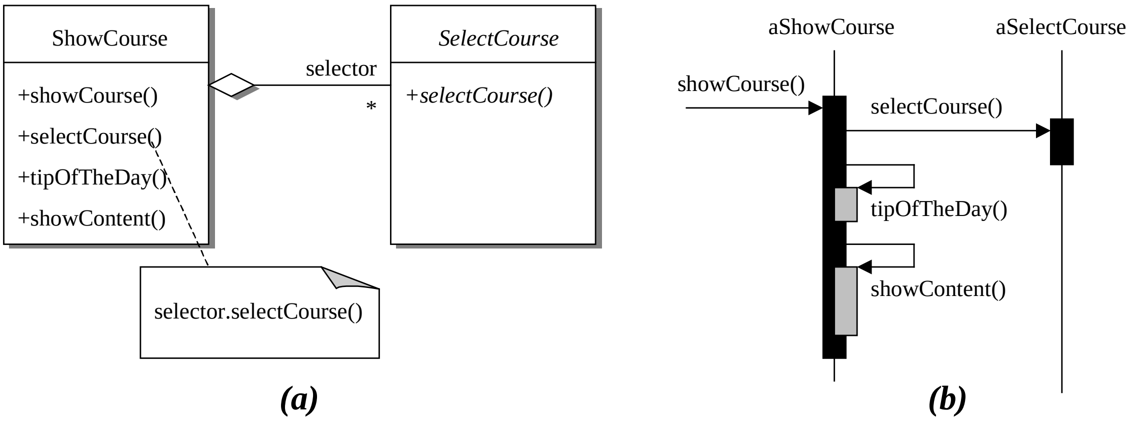

Figure 1. UML representation of a framework

The method *tipOfTheDay()* is also a framework variation point. The reason it that it may be wanted by some instances but may be unwanted by others. The framework should provide only the methods and information that are useful for all possible application instances in its kernel classes. The extra functionality, which may be referred to as should be provided only in the application code. A notation to designate this kind of variation points would be extremely useful in framework design. A design for the class *ShowCourse* could include a potentially infinite number of methods in the class, many of which would not be needed in a given application. Also, the inclusion of methods like *tipOfTheDay()* could result in a complex interface for *ShowCourse*, which is a bad design heuristic [26].

This simple example shows that standard OO modeling notations could be usefully augmented with constructors to distinguish variation points from the framework kernel, since there are several problems that are directly related to this “standard” design representation:

- Variation point modeling: the framework variation points are not first class citizens in the standard design languages. They have to be represented with standard object-oriented constructors, which has two undesirable consequences. The first is the difficulty of identifying the variation points in the framework structure, encumbering the instantiation process. The second is the complexity of the design diagrams, especially in the case that “tricks” are used add flexibility and extensibility to the framework, overcoming the lack of appropriate variation point constructors. Design patterns [2, 12, 30] are examples of “tricks” that might be used;

- Instantiation process: generally, framework instantiation is far more complex than simply plugging components into variation points. The variation points might have interdependencies [8], might require instantiation for every instance or be optional [1, 8], the framework may provide several ways of adding the same functionality [31], and so on. The instantiation step should then be explicitly represented as a process so that the framework user would have no doubt in how to create a valid framework instance.

- Changing the implementation of an existing framework class every time new functionality is needed, as in the case of method *tipsOfTheDay()*, is also not good practice. If these variation point methods could be marked, the person who is creating the application from the framework and the application maintainer would know how they are invoked and which methods could be removed without disturbing the system.

There are basically three kind of variation points in object-oriented frameworks: *variable methods*, *extensible classes*, and *extensible interfaces*. Variable methods are methods that the implementation may vary, depending on the framework instantiation. In the example, method *selectCourse()*, is a variable method. Extensible classes are classes that the interface depends on the framework instantiation, like the *ShowCourse* class, which may require new instance methods *(tipsOfTheDay()* for example) depending on the instantiation. Extensible interfaces are UML interfaces (that can be viewed as a design model for Java interfaces) or abstract classes that require the creation of concrete (sub)classes to provide the desired functionality. The instantiation of this last kind of variation point is made through the creation new classes, which are called *Application Classes* since they exist only in the applications created from the framework.

In the web-based education framework [8] class *Actor* is an extensible interface variation point since new types of actors may be defined depending on a given framework instance. The default actor types are students, teachers, and administrators, however new types may be needed such as monitors, advisors, and secretaries. This design structure is presented in Figure 2.

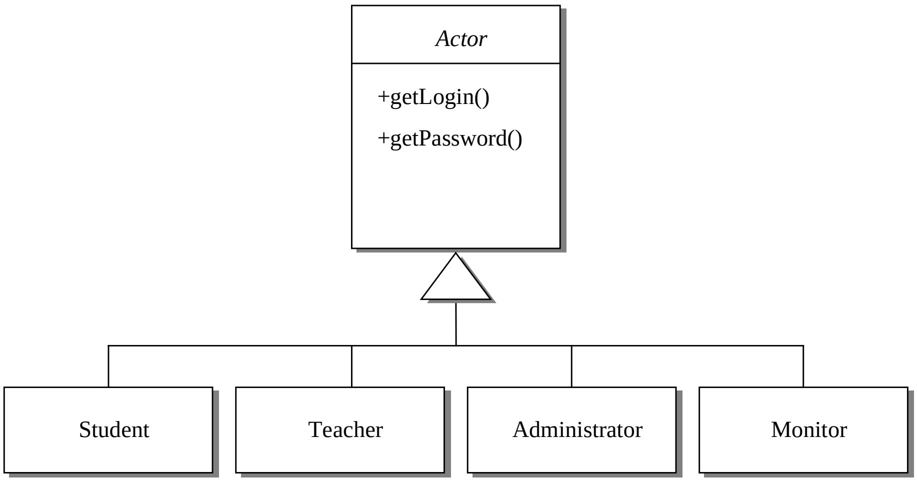

Figure 2. Example of an extension interface variation point

It should be clear that these three kinds of variation points have different semantics: in variable variation points the method implementation varies, in extensible classes the class interface varies, and in extensible interfaces the types in the system vary (new classes are required).

UML provides three language extension mechanisms: stereotypes, tagged values, and constraints. Stereotypes are extensions to the UML vocabulary. They allow the definition of new building blocks that may be used in conjunction with the standard ones. Tagged values are used to extend the properties of a building block. Finally, constraints may be used to modify the semantics of any UML element.

The tagged values *{variable}* and *{extensible}* extend the UML class definitions to identify two different kinds of variation points: variable methods and extensible classes, respectively. In this example *{variable}* refers to *selectCourse()*, indicating that its implementation may vary depending on the framework instantiation; *{extensible}* applies to the *ShowCourse* class, indicating that its interface may be extended during the framework instantiation by the addition of new methods, such as *tipOfTheDay()*.

The *incomplete* constraint is used to identify class *Actor* as an extensible interface, meaning that new subclasses of *Actor* may be defined by framework instances. The stereotype *instance class* indicates classes that are not part of the framework structure and belong to instance applications only. The grayed class *NewActorType* is an example of an instance class. Incomplete allows several instance classes to be created from a given extensible interface during the framework instantiation.

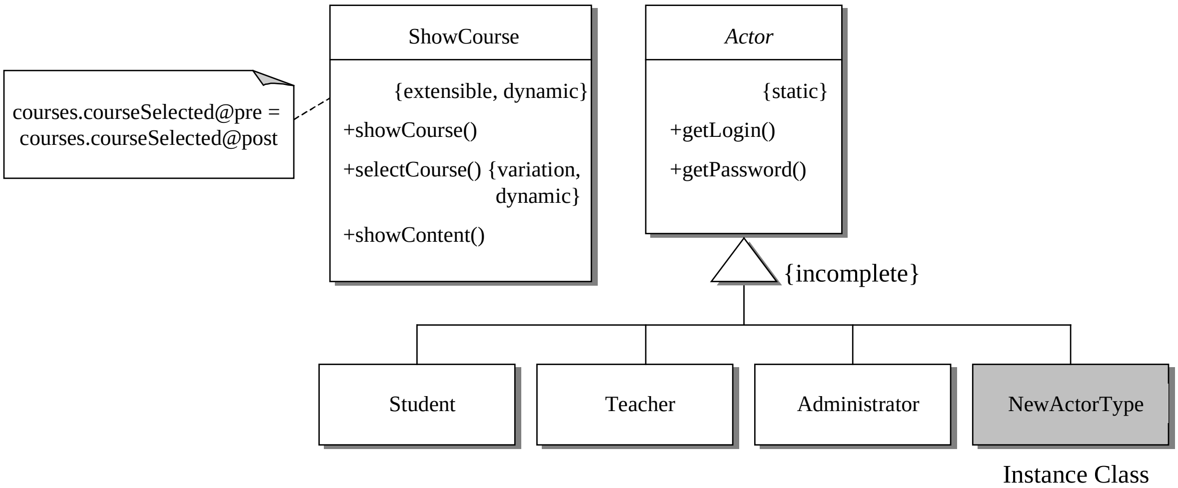

Figure 3. Extended class diagrams

Incomplete is a constraint provided by the standard set of UML constraints, which specifies that not all children of a given generalization relationship have been defined yet. In this work, its meaning was slightly altered to also encompass realization relationships and to assure that the missing subclasses are instance classes that will be completed only during instantiation.

In this example, applications created from the framework always have three kinds of actors, students, teachers, and administrators, but several actor types may be defined depending on the application specific requirements.

The tagged value *{dynamic}* complements the variation point definition by informing that it may require runtime instantiation. Each variation point must be identified either by the *{dynamic}* or by the *{static}* tagged value, where this last one is used to indicate variation points that do not need runtime instantiation. Variable methods are instantiated by providing the method implementation. Extensible classes are instantiated by the addition of new methods. Extensible interfaces are instantiated by the creation of new instance classes.

Table 1 summarizes these elements and informally defines their semantics.

Name of extension	Type of ext-ension	Applies to notational element of UML	Informal semantics
Instance Class	Stereotype	Class	Classes that exist only in the framework instance. New instance classes are defined during the framework instantiation.
Variable	Tagged value	Method	It means that the method implementation depends on the framework instantiation. It is used to identify variable method.
Extensible	Tagged value	Class	It means that the class interface depends on the framework instantiation: new methods may be defined to extend the class functionality. This element identifies extensible classes.
Static	Tagged value	Extensible Interface, Variable Method, and Extensible Class.	It means that the variation point does not require run-time instantiation.
Dynamic	Tagged value	Extensible Interface,	It means that the variation point may require

		Variable Method, and Extensible Class.	run-time instantiation.
Incomplete	Constraint	Generalization and Realization	It has almost the same meaning as in standard UML. Incomplete means that new classes that satisfy a given relationship (generalization or realization) may be added during the framework instantiation, identifying extensible interfaces.

Table 1. Summary of the new elements and their meanings

Note that this design is more abstract than the class diagrams represented in plain UML, since the new constructs cannot be directly mapped into constructs available in existing OO programming languages. The UML standard class diagrams are low-level diagrams that resemble the structure of current OO languages, like Java and Smalltalk.

Figure 4 shows the instantiation diagram for the framework illustrated in the example. It is a representation of the instantiation process that uses the same syntax of the UML activity diagrams. Activity diagrams are used to represent workflow procedures in standard UML [27]. Each action state is used to represent one variation point. The transitions indicate the order variation points should be instantiated.

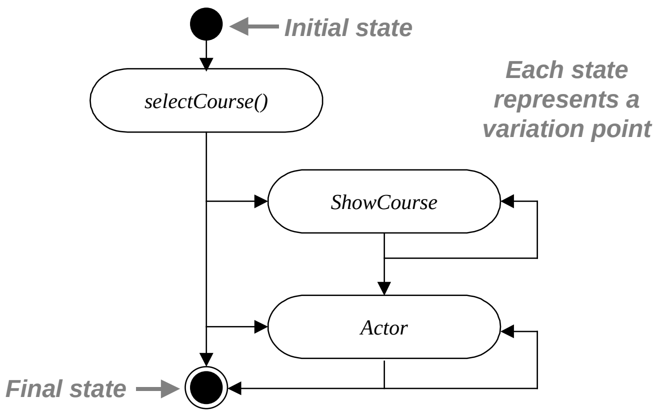

Figure 4. Instantiation diagram

Several facts related to the framework instantiation may be derived from this diagram. The extension of *ShowCourse* interface is optional, since it may not be required by a given framework instance. On the other hand, it may be extended with several methods. The same is true for *Actor*: several *Actor* subclasses may be defined by a given application but none may be needed by another. The *selectCourse()* variable method is not optional and has to be configured by every framework instance. Finally, only one course selection algorithm is allowed per instance.

Extended class diagrams and instantiation diagrams complement each other, and together completely specify variation points and how the framework should be instantiated.

It is very important that framework developers provide documentation that describes what parts of the system should be changed to create a valid framework instance. It is very unlikely that a framework user will be able to browse the framework code, which generally has complex and large class hierarchies, and write the appropriate code if the framework is not well documented. These diagrams address this problem.

2. LOCAL SEARCH HEURISITCS AND THE SEARCHER FRAMEWORK

Hard combinatorial optimization problems usually have to be solved by approximate methods. Constructive methods build up feasible solutions from scratch. Among them we find the so-called greedy algorithms, based on a preliminary ordering of the solution elements according to their cost values. Basic local search methods are based on the evaluation of the neighborhood of successive improving solutions, until no further improvements are possible. As an attempt to escape from local optima, some methods allow controlled deterioration of the solutions in order to diversify the search [1].

In the study of heuristics for combinatorial problems, it is often important to develop and compare, in a systematic way, different algorithms, strategies, and parameters for the same problem. The *Searcher* framework [1] encapsulates different aspects involved in local search heuristics, such as algorithms for the construction of initial solutions, methods for neighborhood generation, and movement selection criteria. Encapsulation and abstraction promote unbiased comparisons between different heuristics, code reuse, and easy extensions.

2.1 SEARCHER PATTERN-BASED DESCRIPTION

This section describes the framework as it was first documented by its authors [1], using a variation of the pattern form proposed in [12] and OMT diagrams [27].

Intent:

To provide an architectural basis for the implementation and comparison of different local search strategies.

Motivation:

In the study of heuristics for combinatorial problems, it is important to develop and compare, in a systematic way, different heuristics for the same problem. It is frequently the case that the best strategy for a specific problem is not a good strategy for another. It follows that, for a given problem, it is often necessary to experiment with different heuristics, using different strategies and parameters.

By modeling the different concerns involved in local search in separate classes, and relating these classes in a framework, our ability to construct and compare heuristics is increased, independently of their implementations. Implementations can easily affect the performance of a new heuristic, for example due to programming language, compiler, and other platform aspects.

Applicability:

The *Searcher* framework can be used in situations involving:

- Local search strategies that can use different methods for the construction of the initial solution, different neighborhood relations, or different movement selection criteria;
- Construction algorithms that utilize subordinate local search heuristics;
- Local search heuristics with dynamic neighborhood models.

Structure:

Figure 5 shows the classes and relationships involved in the *Searcher* framework.

Participants:

- *Client:* contains the combinatorial problem instance to be solved, its initial data and the pre-processing methods to be applied. It also contains the data for creating the *SearchStrategy* that will be used to solve the problem. Generally, it can have methods for processing the solutions obtained by the *SearchStrategy*.
- *Solution:* encapsulates the representation of a solution for the combinatorial problem. It defines the interface the algorithms must use in order to construct and modify a solution. It delegates to *IncrementModel* or to *MovementModel* requests to modify the current solution.
- *SearchStrategy:* constructs and starts the *BuildStrategy* and the *LocalSearch* algorithms, also handling their intercommunication, in case it exists.
- *BuildStrategy:* encapsulates constructive algorithms in concrete subclasses. It investigates and eventually requests *Solution* for modifications in the current solution, based on an *IncrementModel*.
- *LocalSearch:* encapsulates local search algorithms in concrete subclasses. It investigates and eventually requests *Solution* for modifications in the current solution, based on a *MovementModel*.

- *Increment:* groups the data necessary for an atomic modification of the internal representation of a solution for constructive algorithms.
- *Movement:* groups the data necessary for an atomic modification of the internal representation of a solution for local search algorithms.
- *IncrementModel:* modifies a solution according to a *BuildStrategy* request.
- *MovementModel:* modifies a solution according to a *LocalSearch* request.

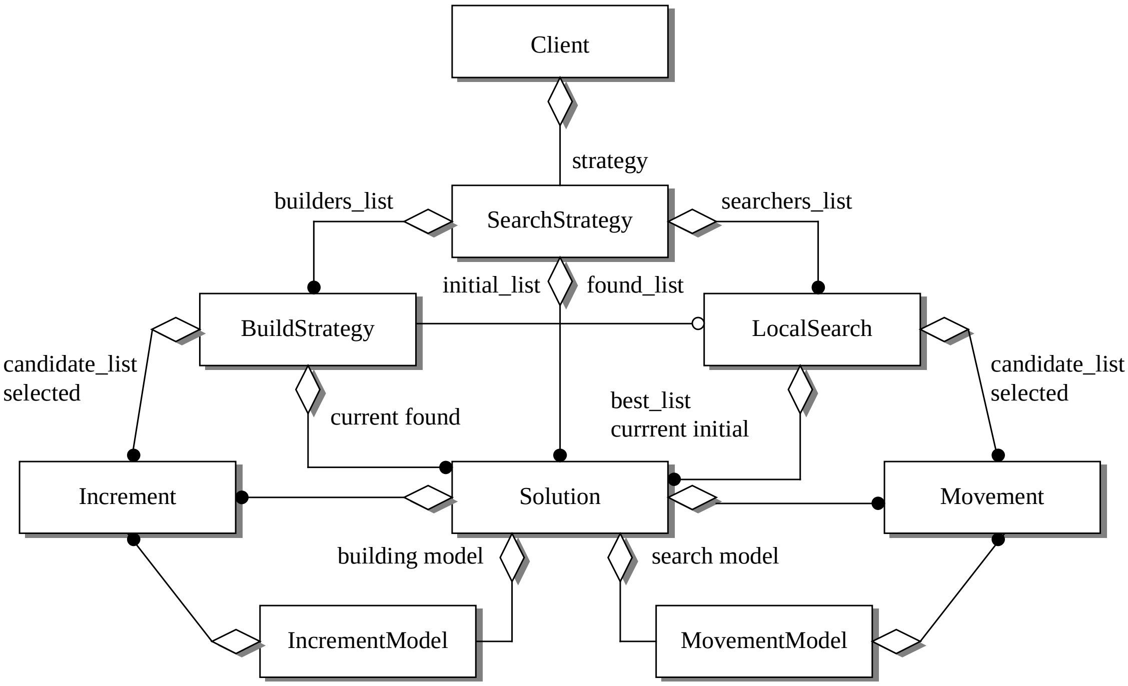

Figure 5. Searcher class diagram

Collaborations:

The *Client* wants a *Solution* for a problem instance. It delegates this task to its *SearchStrategy*, which is composed by at least one *BuildStrategy* and one *LocalSearch.* The *BuildStrategy* produces an initial *Solution* and the *LocalSearch* improves the initial *Solution* through successive movements. The *BuildStrategy* and the *LocalSearch* perform their tasks based on neighborhood relations provided by the *Client*.

The implementation of these neighborhoods is delegated by the *Solution* to its *IncrementModel* (related to the *BuildStrategy*) and to its *MovementModel* (related to the *LocalSearch*). The *IncrementModel* and the *MovementModel* are the objects that will obtain the *Increment*s or the *Movement*s necessary to modify the *Solution* (under construction or not).

The *IncrementModel* and the *MovementModel* may change at run-time, reflecting the use of a dynamic neighborhood in the *LocalSearch*, or having a *BuildStrategy* that uses several kinds of *Increment*s to construct a *Solution*. The variation of the *IncrementModel* is controlled inside the *BuildStrategy* and the variation of the *MovementModel* is controlled by the *LocalSearch*. This control is performed using information made available by the *Client* and accessible to these objects.

Figure 6 illustrates this scenario.

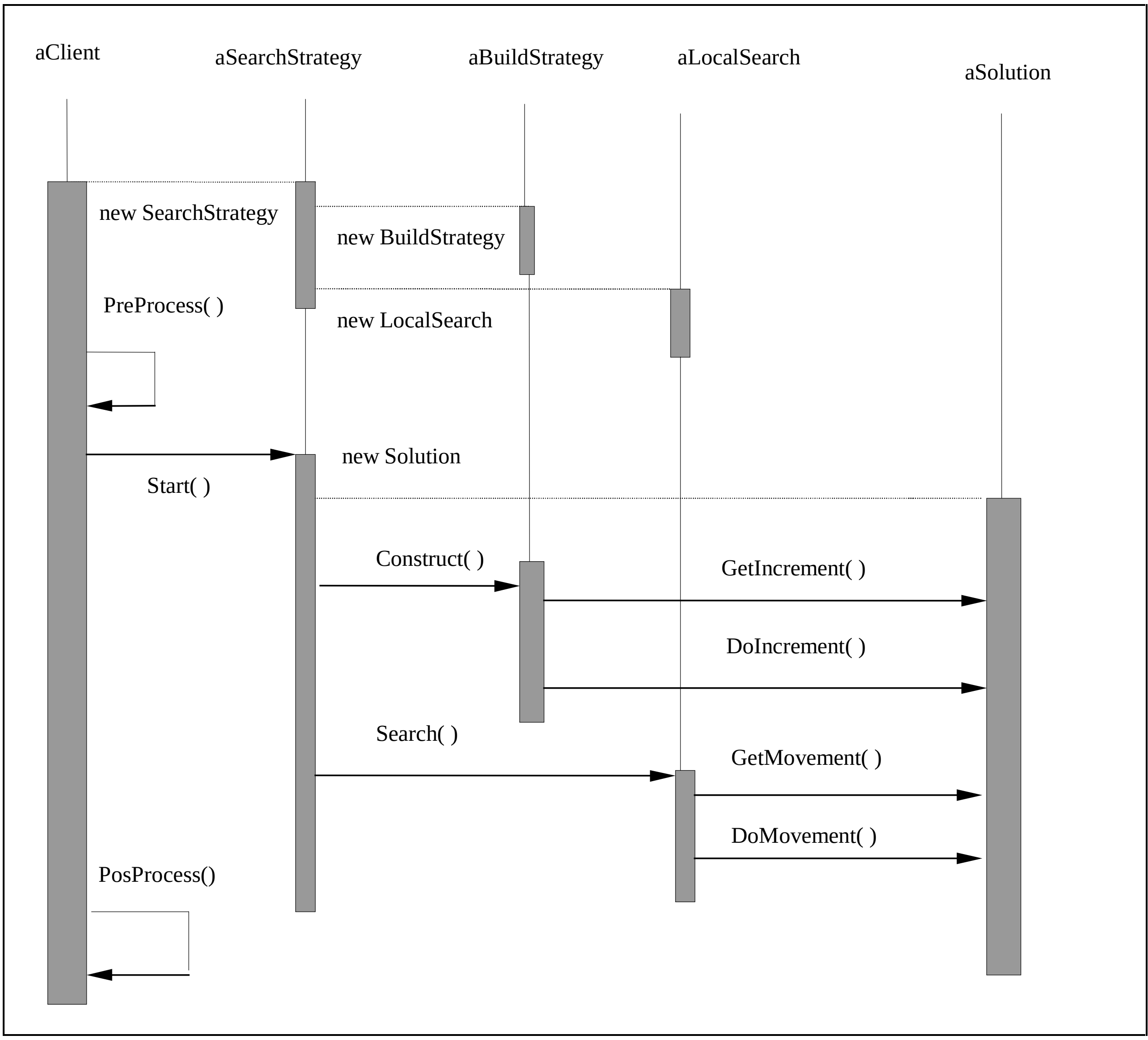

Figure 6. Collaborations in the Searcher framework

3. SEARCHER DESIGN MODEL IN UML-FW

Although the pattern-based description gives an intuition of the framework design structure there are several problems related to it:

- Variation point identification: there is no indication in the diagrams of the framework variation points and how they need to be instantiated. The textual description is informal and might not be clear enough;
- Complex design model: the design diagram presented is very tangled and hard to be understood;
- Interrelated variation points: there are variation point interdependencies that are not represented in the diagrams. Whenever new build strategies are defined new increment models also need to be. The same holds for the search strategies and the movement models.

Figure 7 represents the Searcher design structure through an extended class diagram while Figure 8 illustrates its instantiation diagram. Note that now the variation points are explicitly documented, the design is more clear (four classes have been eliminated) and the variation point interdependencies are formally documented in the instantiation diagram.

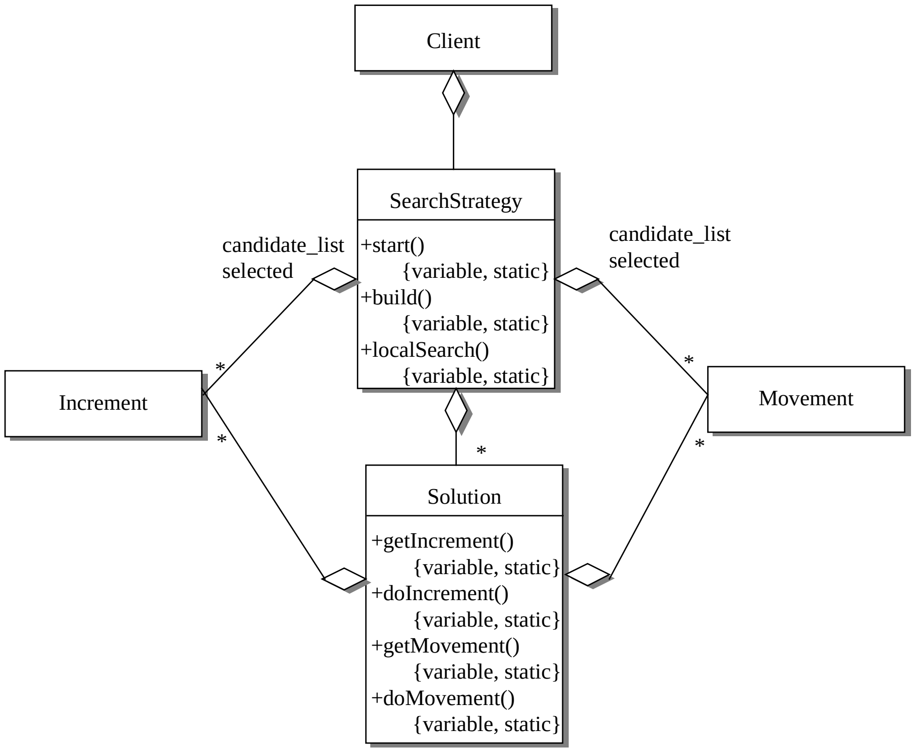

Figure 7. UML extended class diagram for the Searcher framework

The implementation of variation points is one of the most critical parts in framework development. Several techniques such as design patterns [2, 12, 30], meta-level programming [17], and contracts [13, 14] have been proposed to overcome the lack of appropriate constructs to model flexibility and extensibility mechanisms in current OO programming languages. However, the selection of the most appropriate technique for each of the framework's variation points may be a very difficult task. If the hot-spots and their properties are not explicitly represented, this task can become even harder.

Once the variation points have been documented in UML-FW, the application of the most adequate technique to model each variation point may be automated through design transformations. Figure 9 shows a logic program that applies the Strategy design pattern to all variation methods is used in the design diagram shown in Figure 7. The result of this transformation is a diagram almost identical to the one presented in Figure 5. The only difference is that the extension interface variation points are explicitly represented in the generated design, which facilitates the instantiation process since all the places that require the addition of new code are marked in the diagram. The transformation illustrated in Figure 9 is part of the framework development tools proposed in [7], which assist the development and instantation of frameworks using UML-FW.

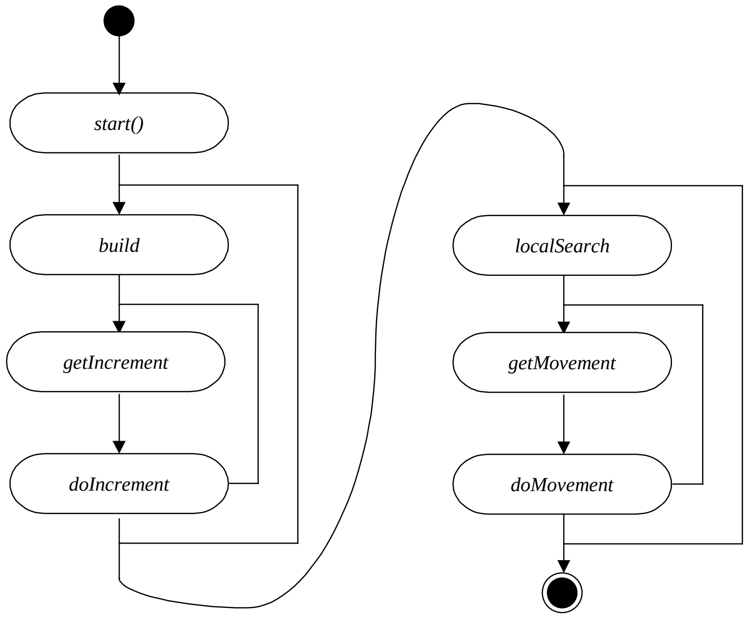

Figure 8. Searcher instantiation diagram

```
applyStrategy(Project, NewProject) :-
          [...]
          forall(variationMethod(Project, Class, Method, dynamic),
          strategy(Project, NewProject, Class, Method)),
          [...]

strategy(Project, NewProject, Class, Method) :-
          concat(Method, 'Strategy', NewClass),
          createClass(NewProject, NewClass, dynamic),
          createMethod(NewProject, NewClass, Method, public, none, abstract),
          createAggregation(NewProject, Class, NewClass, strategy),
          assert(implementation(Project, [...], strategy)).
          [...]
```

Searches for variation hot-spots in a design

Uses strategy pattern to model them

Figure 9. Design-implementation transformation: using Strategy to model variation points

We think that framework builders should not focus on design patterns, but instead on the domain variability and extensibility requirements. Once this knowledge is captured it should be expressed in appropriate design notation, such as UML-FW. Design patterns and other implementation techniques should be considered only in the implementation step of the process, once the design is completely validated by the domain experts. The above example shows that supporting tools can support these ideas by concentrating at the design level and systematizing the mapping from design to implementation.

4. RELATED WORK

UML represents design patterns as collaborations (or mechanisms) and provides a way of instantiating pattern descriptions through the binding stereotype. Frameworks are represented in UML as packages that assemble several patterns [21, 28]. However, framework instantiation can be far more complex than simply assigning abstract classes to concrete ones: new classes and relationship may have to be created, abstract methods have to be implemented, and so on. Catalysis extends the UML approach and proposes a design method for building frameworks [6].

Cookbooks [17] provide a textual description of the purpose of the framework, describing its major components and providing examples of its use. However, cookbooks do not have a formal structure, and different cookbooks may focus on different aspects of the framework with different levels of detail. As well the descriptions are quite informal, and this lack of formality may lead to misconceptions by the user.

Although the patterns described in [16] are based on an Alexandrian narrative, they respect a pattern form and can be seen as a cookbook with more structure.

The hook tool [10, 11] can be seen as a tool to help framework instantiation. The tool uses an extension of UML to frameworks (shading the instance classes) that may help framework design. However it is a very simple extension to UML: it does not support the representation of the variation point and does not classify the variation point types (extension and variation).

The use of role diagrams to represent object collaboration is one of the most promising fields in object oriented design research [3]. Riehle proposes an extension of the OOram methodology [24] to facilitate framework design and documentation [25]. His work proposes a solution for an explicit division of the design, highlighting the interaction of the framework with its clients. The use of roles does simplify the modeling of patterns that require a lot of object collaboration and provides a solution for documenting classes that participate in several design patterns at the same time. However, it does not provides an explicit representation for variation points and instance classes, and does not model the instantiation process.

Adaptable Plug-and-Play Components (APPCs for short) [20] is a language support concept to help the specification of object collaboration diagrams [24] and can help the implementation of frameworks as shown in [7]. Also Lieberherr and the researchers of the Demeter Project [19] have developed a set of concepts and tools to help and evaluate object oriented design that may be applied to design patterns and frameworks.

Some work in the systematic application of patterns to implement framework variation points can be fund in [23, 29].

5. CONCLUSIONS AND FUTURE WORK

The main goal of this research is to define an adequate notation for frameworks that is useful in the implementation and instantiation steps of the software development process. By choosing an appropriate representation at the design step then the subsequent steps of implementation and instantiation can become more straightforward and may even be amenable to partial automation through the use of tools.

The case study described in this paper shows how framework instantiation can be vastly improved by the use of UML-FW notation. Representing frameworks at a higher level of abstraction than programming languages can be very useful to assist with an understanding of a design.

More case studies of the use of the proposed notation to describe and implement real world frameworks and the specification, including the architecture and functionality, of an environment that supports the creation of frameworks using this notation is found in [7]. This environment supports design analysis over framework structure, generates code using various approaches, generates documentation models, and helps the instantiation process. The first version of such an environment is completely developed and was used and validated in the development of several frameworks.

A new version of this tool that provides cooperative work capabilities and graphical representation is now being developed in Java. This tool will be used to experiment with different objet-oriented models and evaluate the impact of these models to framework design. More concretely, the tool will provide an uniform way of validating how roles [24, 25], and ADVs [4] can enhance framework design and how they can be used together with UML-FW.

REFERENCES

1. A. Andreatta, S. Carvalho, and C. Ribeiro, "An Object-Oriented Framework for Local Search Heuristics", 26th TOOLS, *IEEE Press*, 33-45, 1998.

2. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.

3. J. Coplien, "Broadening beyond objects to patterns and other paradigms", *ACM Computing Surveys*, 28(4es), 152-es, 1996.

4. D. Cowan and C. Lucena, “Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse”, *IEEE Transactions on Software Engineering*, 21(3), 229-243, 1995.

5. D. D’Souza, A. Sane, and A. Birchenough, “First-class Extensibility for UML – Packaging of Profiles, Stereotypes, Patterns”, UML’99, *LNCS 1723*, Springer-Verlag, 265-277, 1999.

6. D. D’Souza and A. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison Wesley, 1997.

7. M. Fontoura, “A Systematic Approach for Framework Development“, Ph.D. Thesis, Computer Science Department, PUC-Rio, 1999 (http://www.cs.princeton.edu/~fontoura).

8. M. Fontoura, L. Moura, S. Crespo, and C. Lucena, “ALADIN: An Architecture for Learningware Applications Design and Instantiation”, Technical Report MCC34/98, Computer Science Department, PUC-Rio, 1998 (also submitted to WWW Journal).

9. M. Fontoura, W. Pree, and B. Rumpe, “UML-FW – A Modeling Language for Object-Oriented Frameworks”, Technical Report 613-99, Princeton University, 1999 (also submitted to ECOOP 2000).

10. G. Froehlich, H. Hoover, L. Liu, and P. Sorenson, “Hooking into Object-Oriented Application Frameworks”, ICSE’97, *IEEE Press*, 491-501, 1997.

11. G. Froehlich, H. Hoover, L. Liu, and P. Sorenson, “Requirements for a Hooks Tool”, (http://www.cs.ualberta.ca/~softeng/papers/papers.htm).

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

13. R. Helm, I. Holland, and D. Gangopadhyay, “Contracts: Specifying Behavioral Composition in Object-Oriented Systems”, OOPSLA/ECOOP’98, Norman Meyrowitz (ed.), *ACM Press*, 169-180, 1990.

14. I. Holland, “The Design and Representation of Object-Oriented Components”, Ph.D. Dissertation, Computer Science Department, Northeastern University, 1993.

15. P. Hudak, “Building Domain-Specific Embedded Languages”, *ACM Computing Surveys*, 28(4es), 196-es, 1996.

16. R. Johnson, “ Documenting Frameworks Using Patterns”, OOPSLA’92, *ACM Press*, 63-76, 1992.

17. G. Kiczales, J. des Rivieres, and D. Bobrow, *The Art of Meta-object Protocol*, MIT Press, 1991.

18. G. Krasnes and S. Pope, “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80”, *Journal of Object-Oriented Programming*, 1(3), 26-49, 1988.

19. K. Lieberherr and I. Holland, “ Assuring Good Style for Object-Oriented Programs”, *IEEE Software*, September, 38-48, 1989.

20. M. Mezini and K. Lieberherr, “Adaptative Plug-and-Play Components for Evolutionary Software Development”, OOPSLA’98, *ACM Press*, 97-116, 1998.

21. OMG, “OMG Unified Modeling Language Specification V.1.2”, 1998 (http://www.rational.com/uml).

22. D. Parnas, P. Clements, and D. Weiss, “The Modular Structure of Complex Systems”, *IEEE Transactions on Software Engineering,* SE-11, 259-266, 1985.

23. W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

24. T. Reenskaug, P. Wold, and O. Lehne, *Working with objects,* Manning, 1996.

25. D. Riehle and T. Gross, “Role Model Based Framework Design and Integration”, OOPSLA’98, *ACM Press*, 117-133, 1998.

26. A. Riel, *Object-Oriented Design Heuristics*, Addison-Wesley, 1995.

27. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Clifs, 1994.

28. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998.

29. H. Schmid, “Systematic Framework Design by Generalization”, *Communications of the ACM*, 40(10), 48-51, 1997.

30. J. Vlissides, *Pattern Hatching: Design Patterns Applied*, Software Patterns Series, Addison-Wesley, 1998.

31. J. Vlissides, “Generalized Graphical Object Editing”, Ph.D. Dissertation, Department of Electrical Engineering, Stanford University, 1990.