Extending UML to Improve the Representation of Design Patterns

Marcus Fontoura and Carlos Lucena
Software Engineering Laboratory (LES)
Computer Science Department, Pontifical Catholic University of Rio de Janeiro
Rua Marquês de São Vicente, 225, 22453-900 Rio de Janeiro, Brazil
e-mail: {mafe, lucena}@les.inf.puc-rio.br

ABSTRACT

Several design patterns are defined to make systems more flexible and extensible. The main goal of this work is to show how the representation of this kind of patterns, which we refer to as configuration design patterns, can be vastly improved through extensions to the diagrams used to model them. An extension to the UML design notation to better represent configuration patterns is proposed and illustrated through examples of well-known design patterns and real-world frameworks. The paper also shows that the proposed representation can be more easily mapped to new implementation techniques such as Aspect-Oriented Programming (AOP) and Subject-Oriented Programming (SOP).

KEY WORDS: configuration design patterns, pattern representation, UML, new implementation techniques, frameworks.

1. INTRODUCTION

Configuration design patterns are patterns used to make systems more flexible and extensible. Most of the design patterns proposed in [11] can be classified in this category[1]. This work addresses the problem of configuration design patterns representation. Based on our experience applying design patterns to structure complex frameworks [5, 8, 9] and teaching courses on patterns [6] some conclusions were driven:

- Configuration design patterns need to be instantiated;
- The pattern form used to represent them is based on OOADMs diagrams, such as OMT [23] and UML [24] class and interaction diagrams;
- Current OOADMs do not provide elements and diagrams to represent instantiation, which is a key concept behind configuration patterns;
- Several design patterns are complex design structures that may lead to very tangled diagrams, especially when several patterns are combined in a system.

Based on these premises, we have developed an extension to the UML notation that enhances the representation of configuration patterns by: (i) explicitly representing the pattern instantiation, and (ii) being abstract enough to simplify the diagrams and allow several implementation approaches to be used. The solution was based on the UML extensibility mechanisms [24].

The rest of the paper is organized as follows: Section 2 shows how three well-known design patterns (Strategy, Composite, and Visitor) [11] are represented first in standard UML and then by the proposed notation. This section also compares both representations highlighting the benefits of our approach. Section 3 details the solution describing the syntax and semantics of the new elements. Section 4 shows how the solution can be applied to real-world frameworks that use several configuration patterns. Section 5 describes an approach to map the proposed pattern representation into new upcoming implementation technologies such as AOP [15] and SOP [12]. Section 6 describes related work while Section 7 concludes the paper and outlines our future research directions.

2. EXAMPLES OF DESIGN PATTERN REPRESENTATION

This section presents some well-known patterns using standard UML diagrams, discuss this representation, and

[1] Normally, the Singleton, Adapter, Façade, Flyweight, Interpreter, and Memento design patterns are the only ones from the 23 patterns presented in [11] that will not be classified as configuration patterns. However, this classification is a little bit subtle and depends upon the pattern use.

shows how it can be enhanced by adding new elements to the underlying design notation. Section 3 details the proposed solution.

2.1 STRATEGY

Figure 1 illustrates the structure field of the Strategy design pattern [11] using standard UML. The idea behind this pattern is to encapsulate possible variations of a given algorithm, allowing the system to invoke the more appropriate algorithm depending on a given context.

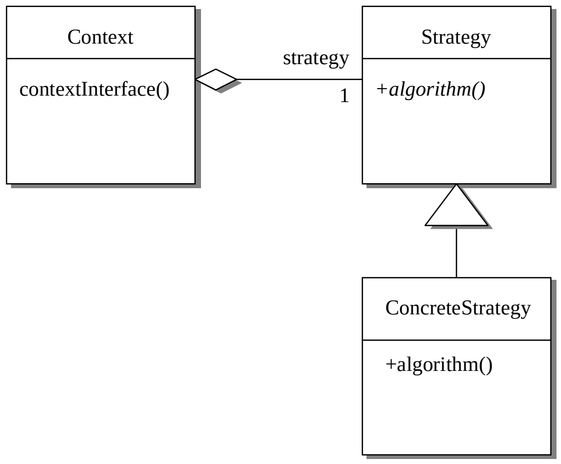


Figure 1. Strategy class diagram in standard UML

The user that wants to use Strategy has to implement the set of possible algorithms in **Strategy** subclasses (**ConcreteStrategy.algorithm()**). This is exactly the pattern instantiation step. The necessity of instantiation step implies that Strategy is a configuration pattern. The understanding of instantiation step is essential for applying Strategy, however Figure 1 does not give any information about it.

Figure 2 shows an extended version of standard UML class diagrams, where the pattern hot-spots [19, 25] are explicitly represented. A hot-spot is defined here as the aspect of the pattern that may vary, depending on the pattern instantiation. In this case, the method **algorithm()** is a hot-spot as indicated by the tagged value **variable**, which means that **algorithm()** implementation may vary. This representation precisely captures the idea behind Strategy, which is to encapsulate algorithm variations. Another difference from this diagram and the standard one presented in Figure 1 is that the class **ConcreteStrategy** is highlighted, indicating that it is an application class. Application classes, differently from the standard ones, exist only after the pattern is instantiated. The highlighted classes are the ones that have to be added to the system to implement a new pattern , and are the ones that the user should focus when configuring the system. Considering the Strategy pattern, several classes may be provided to implement the different algorithms that are used within that pattern .

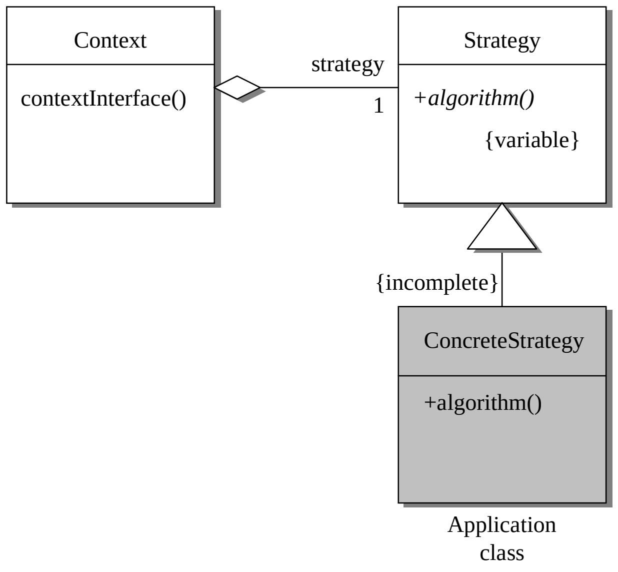


Figure 2. Explicit representation of the pattern hot-spots

The **incomplete** constraint indicates that more subclasses of **Strategy** may be created. **Incomplete** is provided by the standard set of UML constraints [24].

Although Figure 2 is more expressive than the standard one regarding the pattern instantiation, it can still be complemented with the instantiation diagram presented in Figure 3. Currently none of the existent OOADMs provide diagrams that have the similar semantics to the instantiation diagrams described here. The idea is to define the steps that have to be performed during the pattern instantiation in a visual process language. Instantiation diagrams are defined using the elements provided by UML activity diagrams, which are the ones used to represent workflow models.

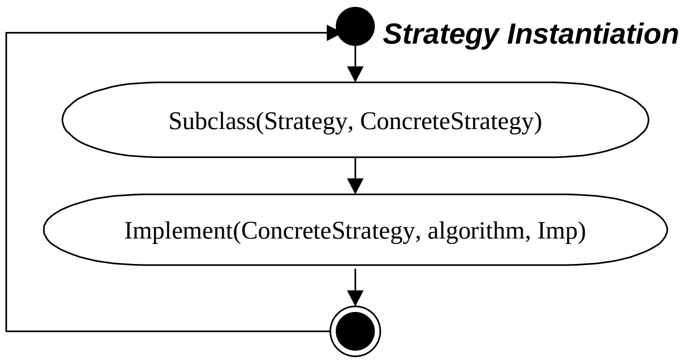

Strategy Instantiation
Subclass(Strategy, ConcreteStrategy)
Implement(ConcreteStrategy, algorithm, Imp)

Figure 3. Instantiation diagram for Strategy

Figure 2 and Figure 3 complement each other and together provide a more expressive representation for the Strategy pattern than the one showed in Figure 1. The following subsections show how other two configuration patterns described in [11] can be better represented using the same concepts. Section 3 generalizes the solution, describing the syntax and semantics of the extensions we made to the UML notation.

2.2 COMPOSITE

Figure 4 illustrates the Composite design pattern [11] class diagram in UML. Composite provides a uniform way to create object trees that represent part-whole hierarchies. The idea behind this pattern is to make composite object are independent form its components. Thus, the pattern allows new kinds of components to be added to the system without disturbing the behavior of the composite objects.

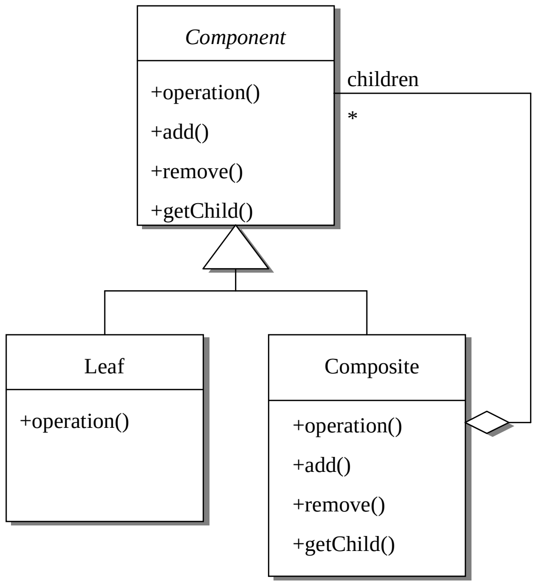


Figure 4. Composite UML class diagram

A user who wants to apply composite has to define the set of components used within the intended pattern . Another important point is that this set of components may evolve during the system lifetime through the addition of new leaf classes. Different instantiations of the pattern may have a different set of leaf classes. Figure 5 and Figure 6 explicitly illustrate this instantiation step.

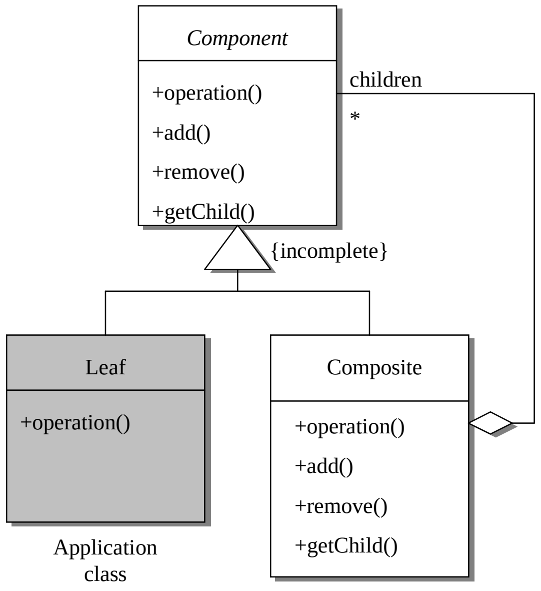


Figure 5. Highlighting the configuration step

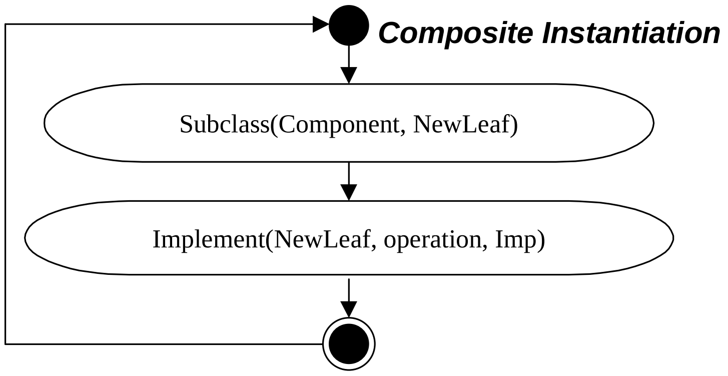

Composite Instantiation
Subclass(Component, NewLeaf)
Implement(NewLeaf, operation, Imp)

Figure 6. Composite pattern instantiation diagram

2.3 VISITOR

This design pattern adds extensibility to a system, allowing the definition of new operations without changing the interface of the class on which they operate. Figure 7 shows its UML class diagram. In order to add new operations to the system a user that applies Visitor should create new classes in the **Visitor** hierarchy and implement the desired operations on the newly created classes.

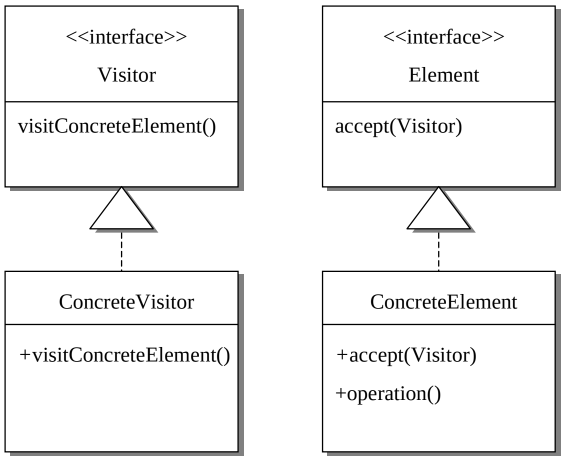


Figure 7. Visitor UML class diagram

Figure 8 extends the standard class diagram by identifying the pattern hot-spot and highlighting the application classes, which are the ones that implement the **Visitor** interface. The hot-spot is identified by **extensible** tagged value in the method container of the **ConcreteElement** class. This representation indicates that new methods may be added to the element classes[2]. The applicability of the **incomplete** constraint is enlarged here to also encompass the realization relationship, meaning that new classes that implement a given interface (Visitor in this case) may be added to the project during instantiation time. The corresponding instantiation diagram is presented in Figure 9.

[2] This tagged value is just representing the semantics of the pattern, since in the Visitor implementation new methods are not added to **ConcreteElement** classes directly. However, the idea behind the pattern is to "extend" the class interface and the "extensible" tagged value documents that precisely.

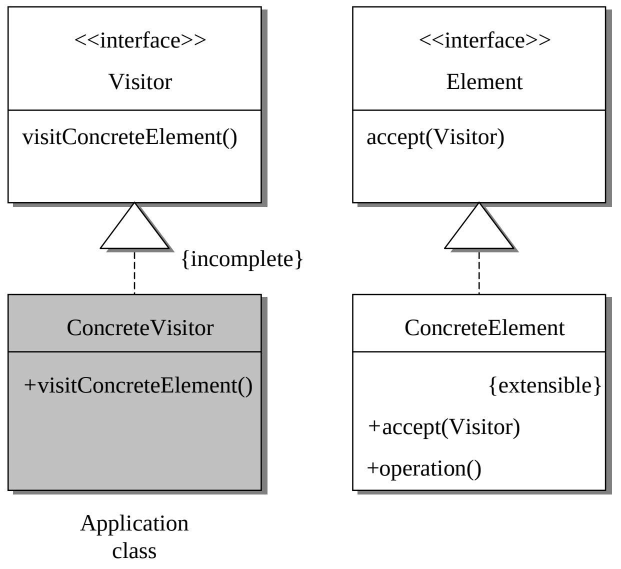
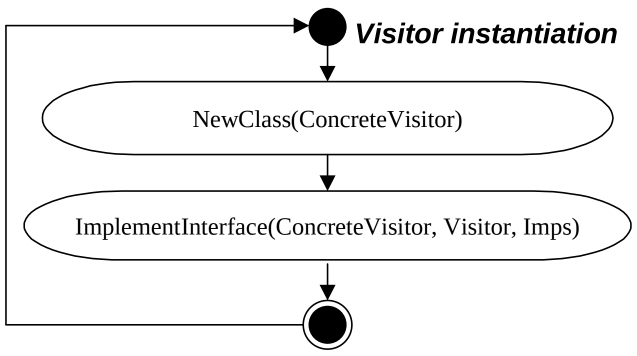


Figure 8. Identifying Visitor hot-spots



Visitor instantiation
NewClass(ConcreteVisitor)
ImplementInterface(ConcreteVisitor, Visitor, Imps)

Figure 9. Visitor pattern instantiation diagram

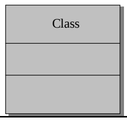
3. PROPOSED SOLUTION

As shown by the design patterns described in section 2, the proposed solution for enhancing the configuration patterns representation extends UML by:

- Changing the class diagrams to explicitly identify pattern hot-spots and application classes;
- Defining instantiation diagrams, which represent the pattern instantiation process.

To extent UML class diagrams a new stereotype was created, **Application class**, and two new tagged values were defined, **variable** and **extensible**. Finally, the applicability of the **incomplete** constraint was enlarged to encompass dependency relationships as well. Table 1 summarizes these elements and defines their semantics.

The instantiation diagrams are a visual representation of a process language. In the case of isolated patterns the instantiation diagrams tent to be simple, as the ones showed in section 2, however for frameworks that combine several patterns and have special domain specific requirements these diagrams can become quite complex, as will be shown in section 4.

Element	Type	Applies to	Semantics
Application class 	Stereotype	Class	Classes that exist only in the pattern . Generally, application classes model the varying concept encapsulated by the pattern. New application classes are defined during the pattern instantiation.

Variable	Tagged value	Method	Means that the method implementation is the varying concept that the pattern encapsulates. Or in other words, the method implementation depends on the pattern instantiation.
Extensible	Tagged value	Method and Attribute Containers	Means that the class interface (methods, attributes, or both) is varying concept that the pattern encapsulates. Or in other words, the class interface depends on the pattern instantiation: new methods and attributes may be defined to extend the class functionality.
Incomplete	Constraint	Generalization and Realization	Almost the same meaning that in standard UML but applies also to dependency relationships. Incomplete means that new classes that satisfy a given relationship (generalization or realization) may be added during the pattern instantiation

Table 1. Summary of the new elements and their meaning

Instantiation diagrams are represented using the syntax defined by the UML activity diagrams, where each action state defines a transformation over the pattern design [2]. The syntax of the underlying transformational language is PROLOG-like and its semantic is quite intuitive. Some transformation examples are:

- Subclass(Superclass, NewClass): NewClass is a new class added to the system that is a subclass of Superclass. The user has to specify the name of newClass;
- Implement(Class, Method, Imp): asks the user to provide a concrete implementation Imp for the method Method of class Class;
- NewClass(Class): creates a new class in the system. The pattern user has to provide the class name.

The complete language is a variation of the one proposed in [2] and can be found in [8].

4. DOCUMENTING FRAMEWORKS

This section presents two case studies that show how frameworks that assemble several configuration patterns can also be better documented by the proposed UML-extended notation.

4.1 WEB-BASED EDUCATION FRAMEWORK

ALADIN [9] is a web-based education framework that is currently being used to support the development of Web-based applications, such as AulaNet [5] (http://aries8.uwaterloo.ca/aulanet) and OwlNet [1]. We estimate that the use of ALADIN increases productivity by a factor of 3.

Figure 10 illustrates a design model for part of the ALADIN framework. It represents a student subsystem, in which the user has to select the desired course (method **selectCourse**) before he can browse its content, which is displayed by method **showContent**. Figure 10 uses the UML extended representation to explicitly represent the framework hot-spots:

- Method **selectCourse** is a variation hot-spot, which means that several selection mechanisms may be implemented within the framework. This means that each of ALADIN must define its own course selection mechanism. This hot-spot was implemented by the Strategy design pattern;
- Class **ShowCourse** might have its interface extended by the addition of new methods. This allows different s of ALADIN to configure its own set of displaying methods, avoiding defining in the framework classes methods that might be unwanted for some framework s. This hot-spot was implemented by the Visitor design pattern. **TipOfTheDay**, which shows start-up tips, might be a feature wanted by some framework s, as shown in Figure 10.

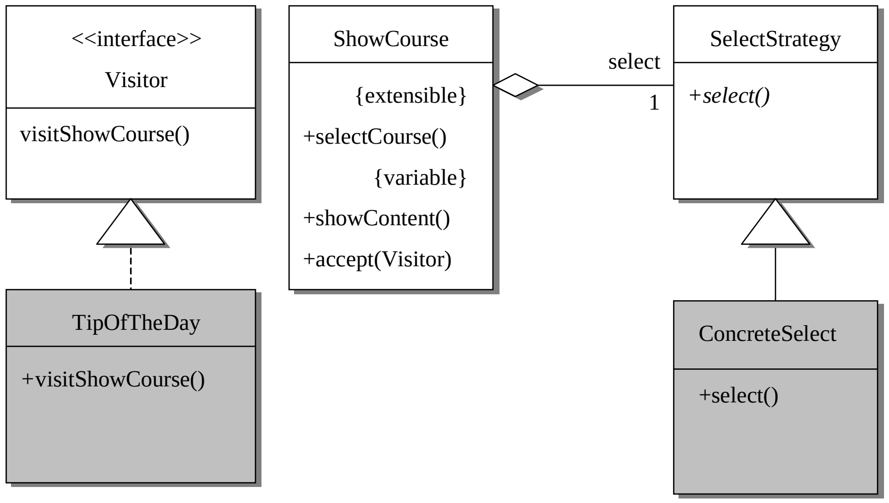
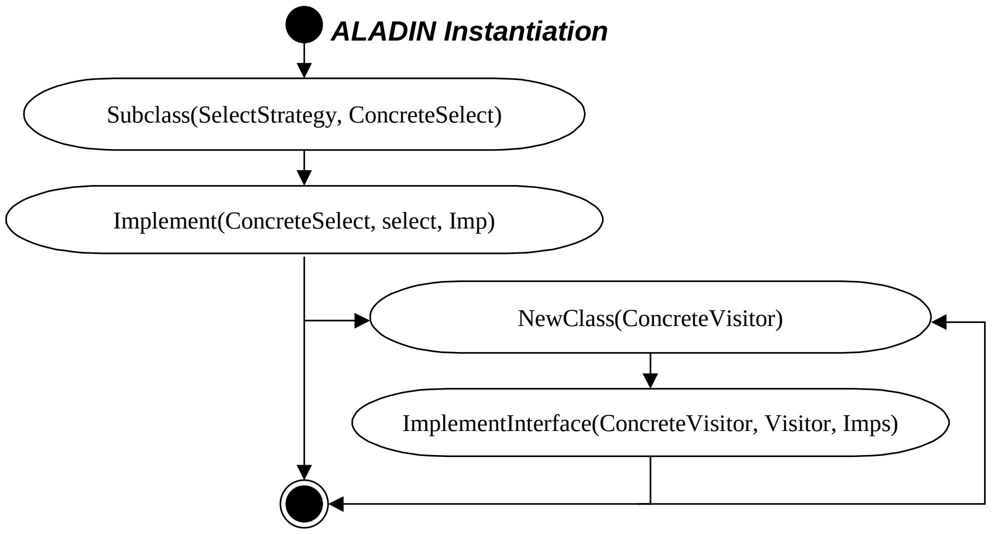


Figure 10. Modeling frameworks

Figure 10 is representing the framework architecture in a more expressive way than if standard UML was used. The identification of the hot–spots by the tagged values **extensible** and **variable** indicate what are and the exact meaning of each of the hot-spots. The application classes indicate what parts of the system are used to instantiate the hot-spots.

Figure 11 presents the instantiation diagram that models how the ALADIN should be instantiated, considering only the portion of the framework described in the example. Note that the extension of **ShowCourse** interface is optional, since it may not be required by a given framework . On the other hand, another may want to extend **ShowCourse** with several methods. The **selectCourse** hot-spot has to be configured by every framework and only one selection algorithm is allowed per .



ALADIN Instantiation
Subclass(SelectStrategy, ConcreteSelect)
Implement(ConcreteSelect, select, Imp)
NewClass(ConcreteVisitor)
ImplementInterface(ConcreteVisitor, Visitor, Imps)

Figure 11. ALADIN instantiation process

Figure 10 and Figure 11 complement each other, and together they completely specify the ALADIN configuration phase. It is very important that framework developers provide documentation that describes what parts of the system should be changed to create a valid framework . 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.

4.2 UNIDRAW FRAMEWORK

Unidraw [26] is a graphical editor framework that allows the construction of domain-specific editors. Different domain-specific editors normally require new graphical components. Unidraw allows the definition of these new components by the creation of composite components (GraphicComp subclasses) from the set of primitive components (Graphic subclasses) and composite components previously defined in the system.

Figure 12 illustrates this design structure. During instantiation, new subclasses of **Graphic** and **GraphicComp** may be defined. **Graphic** subclasses implement the framework primitive components while **GraphicComp** subclasses define composite components. In this example, the composite components are AND, OR, and NAND gates used to model electrical circuits in schematic capture systems.

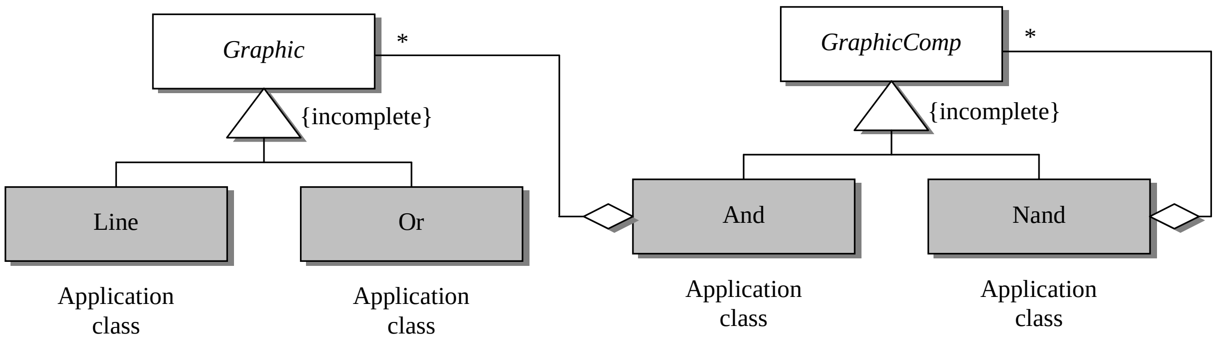


Figure 12. Unidraw Graphic-GraphicComp design structure

However, there is a balance between reuse and runtime performance that the Unidraw user should take in account when defining a new graphic component [27].

1. If he wants maximal reuse, a new component might be defined as a composition of existing GraphicComp subclass. The **Nand** component illustrated in Figure 12 uses this approach. This option may have performance problems due to the visualization approach adopted by the framework;
2. If he wants maximal performance, a new component should be described as a new primitive component (Graphic subclass). The **Or** component showed in Figure 12 uses this approach. This option will require much more implementation effort, since nothing is being reused;
3. An intermediate solution is the definition of a new component by a custom composition of existing Graphic subclasses, as the case of the **And** component (Figure 12).

The instantiation diagram showed in Figure 13 formally presents these instantiation options to the framework user. Note that an application builder that guides the user in the instantiation of new Unidraw applications may be defined based on these diagrams. We are now investigating how builders can be derived from the framework documentation [8].

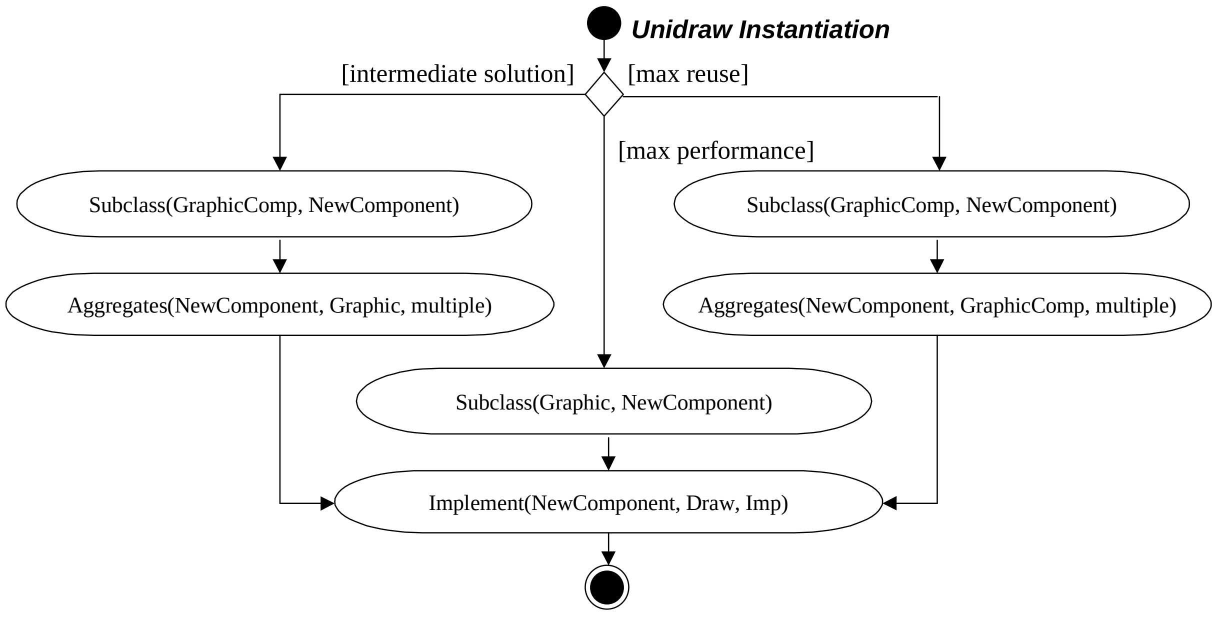

Unidraw Instantiation
[intermediate solution]
[max reuse]
[max performance]
Subclass(GraphicComp, NewComponent)
Aggregates(NewComponent, Graphic, multiple)
Subclass(GraphicComp, NewComponent)
Aggregates(NewComponent, GraphicComp, multiple)
Subclass(Graphic, NewComponent)
Implement(NewComponent, Draw, Imp)

Figure 13. Unidraw Graphic-GraphicComp instantiation diagram

5. NEW IMPLEMENTATION TECHNIQUES

This section shows how the explicit representation of configuration patterns can enhance the implementation step of the development process. Two implementation techniques will be considered: Aspect and Subject-Oriented Programming.

5.1 ASPECT ORIENTED PROGRAMMING

"Aspects" are cross-cutting[3] non functional constraints on a system, such as error handling and performance optimization. Current programming languages fail to provide good support for specifying "aspects," and so code that implements them is typically very tangled and spread throughout the entire system. Aspect-oriented programming (AOP) [15] is a technique proposed to address this problem. An application that is based on the AOP paradigm has the following parts: (i.a) a component language, used to program the system components, (i.b) one or more aspect languages, used to program the aspects, (ii) an aspect "weaver", which is responsible for combing the component and the aspect languages, (iii.a) a component program that implements the system functionality using the component language, and (iii.b) one ore more aspect programs that implement the aspects using the aspect languages.

As discussed throughout the paper, the main idea behind configuration patterns is not the specification of aspects, but the specification of the variability and extensibility requirements of a system. However, AspectJ [17], which is an AOP extension for Java, can be seen as a general-purpose development tool that allows the definition of the program and its aspects in Java.

AspectJ allows the addition of code before or after a method or a constructor is executed (keywords **before** and **after**). The language also allows the addition of code using **catch** and **finally** (similar to Java's catch and finally constructs). All of these keywords determine the points in the component program in which the aspects code written in Java, should execute. AspectJ also provides the **new** constructor, for extending classes with new elements specified in separate aspects.

Even though the primary concern of AspectJ is the specification of non-functional aspects, such as code optimization, it can be used in a straightforward way for implementing configuration patterns.

Figure 14 is an example of how AspectJ can be used to implement the Strategy and Visitor design patterns in the ALADIN framework (described in section 4.1). Aspect **TipOfTheDay** implements a method **showTip**, which is introduced (keyword **new**) to the **ShowCourse** class whenever this aspect is plugged into the system. This provides a clean implementation to the Visitor pattern. Aspects are plugged in by invoking the weaver as shown next.

```
% ajweaver ShowCourse.ajava TipOfTheDay.ajava SelectCourseOption2.ajava
```

```
aspect TipOfTheDay {
            static new void ShowCourse.showTip() { // implementation }
            }
aspect SelectCourseOption1 {
            static after void ShowCourse(*) { // implementation of option 1}
}
aspect SelectCourseOption2 {
            static after void ShowCourse(*) { // implementation of option 2}
}
```

Figure 14. Using AspectJ to implement configuratin patterns

The approach the Strategy pattern is similar: all the different implementations of a given variation are placed in different aspects (**SelectCourseOption1** and **SelectCourseOption2**). When instantiating the framework, one aspect that implements each variation must be plugged-in. In this example, the variation method will execute right after the **ShowCourse** class constructor executes.

[3] If there are two concepts that are better represented in different programming languages (like code optimization and the logic of the system itself) they are said to be cross-cutting concepts [15].

The limitation of this approach to implement configuration patterns is technological: the current implementation of AspectJ is a beta version, and the approach has not yet been used in large scale applications. However, it is important to note that all the information required for implementing and instantiating the aspects can be derived from the UML-extended class diagrams (which is not true for standard class diagrams).

5.2 SUBJECT ORIENTED PROGRAMMING

Subject-oriented programming (SOP) [12] is a paradigm that allows the decomposition of a system into various subjects. A subject compiler [14] can then be used to generate an application by composing the desired subjects. The composition is specified through composition rules such as **Merge** and **Override** [14].

The application of SOP implementation techniques to configuration patterns leads to an implementation very similar to the one presented for AspectJ. A subject would be used to represent each different implementation for each framework hot-spot, and another subject would be used to represent the pattern stable (non-varying) parts. The subjects would then be combined through the use of appropriate composition rules to generate the pattern .

Figure 15 illustrates the approach, where **Framework** is the instantiation of the **selectCourse** hot-spot with the method defined in the subject **SelectCourseOptionX**. Like AOP, SOP is still experimental and there are no industrial subject compilers.

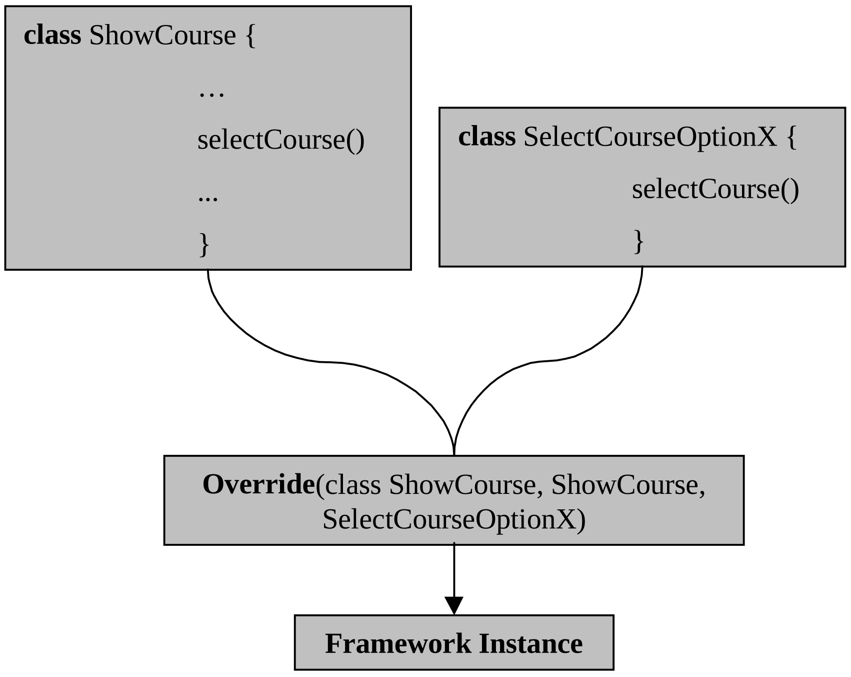

class ShowCourse {
…
selectCourse()
...
}
class SelectCourseOptionX {
selectCourse()
}
Override(class ShowCourse, ShowCourse, SelectCourseOptionX)
Framework Instance

Figure 15. Using SOP to implement the Strategy pattern in ALADIN

The important point here is to note that a more appropriate representation for configuration patterns may also enhance its implementation, perhaps allowing the construction of tools to completely systematize the process.

6. RELATED WORK

UML represents design patterns as collaborations (or mechanisms) and provides a way of instantiating pattern descriptions through the **binding** stereotype [24]. However, pattern 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 to frameworks ands proposes a design method for building frameworks [7].

The use of role diagrams to represent object collaboration is one of the most promising fields in object oriented design research [4]. Riehle proposes an extension of the OOram methodology [21] to facilitate framework design and documentation [22]. 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 hot-spots and application classes, and does not model the instantiation process.

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

Adaptable Plug-and-Play Components (APPCs for short) [18] is a language support concept to help the specification of object collaboration diagrams [21, 22] and can help the implementation of frameworks as shown in [8]. Also Lieberherr and the researchers of the Demeter Project [16] 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 hot-spots can be fund in [20, 25].

7. CONCLUSIONS AND FUTURE WORK

The main goal of this research is to define an adequate representation for patterns and frameworks that is useful in the documentation, implementation, and instantiation steps of the software development process. The proposed representation is complementary to existing OOADMs, and is defined an extension to UML.

This paper presented the definition of configuration patterns and described how their representation can be vastly enhanced with a more appropriate notation. Examples throughout the paper have shown that the approach is also valid to frameworks that assemble several configuration patterns.

More detailed 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 [8]. This environment supports design analysis over framework structure, generates code using various approaches, and generates documentation models. The first version of such an environment is completely developed and was used and validated in the development of several frameworks.

In addition, the use of domain-specific languages (DSLs) [13] and builders to help framework instantiation is being further investigated, where our major goal is the derivation of the DSLs from specifications written in our UML-extended notation. This derivation is already partially supported in the current implementation of our environment.

REFERENCES

1. P. Alencar, D. Cowan, S. Crespo, M. F. Fontoura, and C. J. Lucena, “OwlNet: An Object-Oriented Environment for WBE”, Second Argentine Symposium on Object-Orientation (ASOO’98), 91-100, 1998.
2. P. Alencar, D. Cowan, J. Dong, and C. J. Lucena, “A Transformational Process-Based Formal Approach to Object-Oriented Design”, Formal Methods Europe (FME’97), 1997.
3. E. Casais, “An incremental class reorganization approach”, ECOOP’92, *LNCS 615*, 114-132, 1992.
4. Jim Coplien, “Broadening beyond objects to patterns and other paradigms”, *ACM Computing Surveys*, 28(4es), 1996.
5. S. Crespo, M. F. Fontoura, and C. J. Lucena, “AulaNet: An Object-Oriented Environment for Web-based Education”, International Conference of the Learning Sciences (ICLS’98), 1998.
6. S. Crespo, M. F. Fontoura, and C. J. Lucena, “Object-Oriented Design Course”, Computer Science Department, Pontifical Catholic University of Rio de Janeiro, http://ead.les.inf.puc-rio.br/aulanet (in Portuguese – “Projeto de Sistemas de Software”).
7. D. D’Souza and A. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach,* Addison Wesley, 1997.
8. M. F. Fontoura “A Systematic Approach for Framework Development”, Ph.D. Dissertation, Computer Science Department, PUC-Rio, 1999.
9. M. F. Fontoura, L. M. Moura, S. Crespo, and C. J. Lucena, “ALADIN: An Architecture for Learningware Applications Design and Instantiation”, MCC34/98, Computer Science Department, PUC-Rio, 1998.
10. G. Froehlich, H. Hoover, L. Liu, and P. Sorenson, “Hooking into Object-Oriented Application Frameworks”, ICSE’97, 491-501, 1997.
11. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

12. W. Harrison and H. Ossher, “Subject-Oriented Programming (A Critique of Pure Objects)”, OOPSLA’93, *ACM Press*, 411-428, 1993.

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

14. M. Kaplan, H. Ossher, W. Harrison, and V. Kruskal, “Subject-Oriented design and the Watson Subject Compiler”, OOPSLA’96 Subjectivity Workshop, 1996 (http://www.research.ibm.com/sop/).

15. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, “Aspect-Oriented Programming”, ECOOP’96, *LNCS 1241*, 220-242, 1997.

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

17. C. Lopes and G. Kiczales, “Recent Developments in AspectJ”, ECOOP'98 Workshop Reader, *LNCS 1543*, 1998.

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

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

20. W. Pree, *Framework Patterns*, Sigs Management Briefings, 1996.

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

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

23. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Clifs, 1991.

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

25. H. A. Schmid, “Systematic Framework Design by Generalization”, *Communications of the ACM*, 40(10), 1997

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

27. J. Vlissides, Personal communication, April 1999.