JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 2, March-April 2006

Object-Oriented Design Patterns for Detailed Design

W. Al-Ahmad, NYIT - Bahrain Campus, School of Engineering and Technology, Manama, Bahrain

Abstract

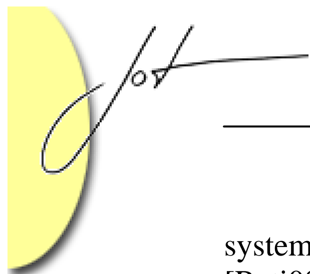
This paper discusses the use of design patterns during the transition phase from analysis to design of object-oriented systems. Pattern mining, which is the process of finding and documenting design patterns, usually takes place during or after the development of a software system. This paper introduces several new design patterns that were discovered through the experience of teaching object-oriented analysis and design in both industry and academia. These are low level design patterns that can be used in almost every application. Finally, a tool will be proposed to automate the application of these patterns when creating a detailed design model.

1 INTRODUCTION

There is increasing pressure on software developers to produce quality software in as short a time as possible. This necessitates the reuse of previously developed or commercially available software elements to expedite the development process. The most common form of reuse is the reuse of code in a fine-grain manner such as objects in the object-oriented paradigm or a large-grain manner such as components in the component-oriented paradigm. With the advent of design patterns [Gamm95] and the pattern-oriented software development methodologies, techniques, and principles [Busc01, Fowl97, Lee03, Shal02, Yaco04], the reuse of designs became possible. Reuse of designs is not as popular as reuse of code because the pattern-oriented approach for software development is still in its infancy. Moreover, constructing generic designs to be later instantiated is a complex issue. There is still a long way to go before pattern-oriented design becomes mainstream with fully-fledged methodologies, tools, and principles.

Usually, the software development life cycle includes the phases of analysis and design regardless of the methodology used (waterfall, iterative, etc.) in the development of the software. This paper follows the Rational Unified Process (RUP) [Rati00] for software development and uses its terminology. RUP is an object-oriented software engineering process that is iterative and incremental. In RUP, analysis and design is one discipline that aims at transforming the requirements of the system into a design of the

Cite this article as follows: W. Al-Ahmad: “Object-Oriented Design Patterns for Detailed Design”, in *Journal of Object Technology*, vol. 5, no. 2, March-April 2006, pp.155-169
http://www.jot.fm/issues/issue_2006_03/article3



system to be built and to adapt the design to match the implementation environment [Rati00]. Understanding and documenting the requirements of the system is the purpose of the requirements discipline, which precedes the analysis and design activities.

Although the line between analysis and design is blurred, there are activities that belong to analysis and other activities that belong to design. Identifying key abstractions of the problem domain and creating a conceptual model are analysis activities, while refining the conceptual (analysis) model by adding more information to it (adding attributes, mapping class responsibility to operations, etc.) are design activities. Refining association multiplicity, adding extra operations related to nonfunctional requirements, optimizing the design model to easily map it to the target language are required to create a detailed design. The focus of this paper is on mining design patterns that recur during these activities. Whereas the majority of the design patterns in the literature focus on high level process activities [Busc01, Gamm95, Larm02, Shal02], the patterns introduced in this paper are low level patterns, focusing on low level design activities. In fact, the patterns introduced in this article can be used in detailing the high-level design patterns.

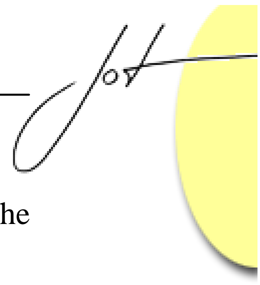
This paper is structured as follows: section 2 briefly explains the problem statement and introduces a conceptual model to model the problem domain. This conceptual model will form the context of the discovered patterns. Section 3 introduces the concept of design patterns and their importance in the development of good design models. Section 4 discusses the new patterns and presents a catalog of these patterns. Section 5 discusses a tool support for automating the incorporation of these patterns into a detailed design model. Finally, section 6 introduces some conclusions and glimpses about future work.

2 CASE STUDY: STUDENT REGISTRATION SYSTEM

This paper uses an on-going example to create a context for the different patterns that will be mined from it. This example is related to the development of a student registration system for a university [Rati00]. To better understand the design models that will be presented in this paper, the relevant part of the problem statement is stated below:

> *A student registration system is needed that will allow students to register for courses and view report cards. Professors will be able to access the system to sign up to teach courses as well as record grades. The system will allow students to select four course offerings for the coming semester. In addition, each student will indicate two alternative choices in case the student cannot be assigned to a primary selection. For each semester, there is a period of time that students can change their schedule by adding or dropping courses [Quat00].*

It is not the objective of this paper to explain the different activities involved to produce the analysis model shown in Figure 1. At the end of the Use Case Analysis activity in RUP, a VOP (View of Participating Classes) or a conceptual model is produced. The VOP is basically a UML class diagram. In Figure 1, the VOP is limited to the key abstractions in the problem domain (entity classes in RUP terminology); it is just a subset of the actual conceptual model produced for the problem at hand. The control and



boundary classes are not shown in the diagram since the patterns are independent of the type of class involved.

Conceptual models and design models should not be developed independently. In RUP, the analysis model evolves to become the design model; the two models are not independent or separate. Object-oriented analysis is supposed to model the problem space; object-oriented design then models the solution space. In the current state of the art, the transition from a conceptual model into a design model is done manually. Typically, people first develop a conceptual model, and then start developing a design model from scratch without a proper binding between both models. In fact, the analysis phase is often skipped in practice, or only a brief study of the problem domain is carried out before turning to the design. This is an inappropriate approach to software development. This way of working does not address the evolutionary nature of the software life-cycle because of problems in traceability between the two phases.

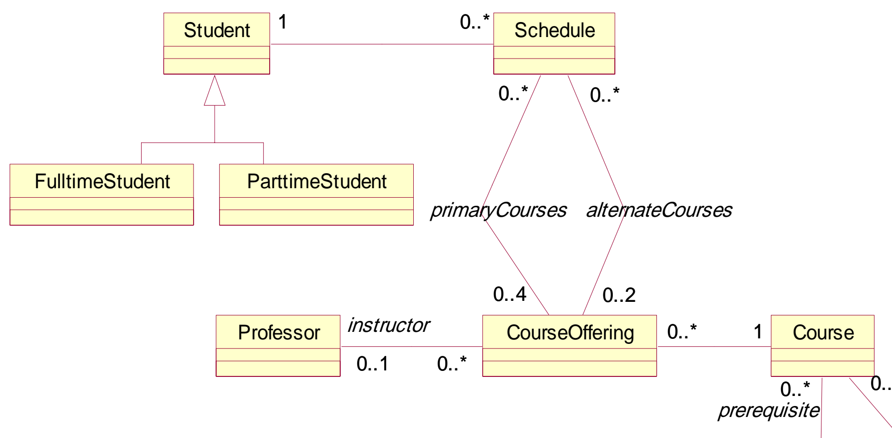


Figure 1: Partial Analysis Model for the Registration System

3 PATTERN-ORIENTED DESIGN

Before we go any further, it is imperative to first define the concept of a design pattern. The GoF book [Gamm95] defines design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." A design pattern names, abstracts, and identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and their instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when the pattern applies, whether or not it can be applied in view of other design constraints, and the consequences and trade-offs of its use. Since we must eventually implement our designs, a design pattern also provides a sample code to illustrate an implementation. The goal of patterns

within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all aspects of software development.

Patterns for software development are one of the latest emerging approaches for software design. Recent work [Yaco04] has recognized the importance of object-oriented design patterns and introduced a methodology for software development with patterns. This paper views object-oriented design as a pattern-based software process in which different types of patterns are successively applied to software artifacts. Each pattern captures a solution to a common object-oriented design problem that is imposed by a particular structure.

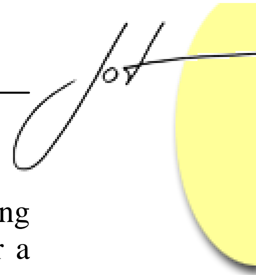
In the literature, there are different description formats for documenting design patterns [Gamm95, Larm02, Shal02]. The description of a design pattern might include items such as the name, problem addressed, context or applicability, forces, solution, structure, etc. In this paper, the description will be limited to the 4 basic elements: context-problem-solution-structure:

- Context: the preconditions under which the problem and its solution seem to recur, and for which the solution is desirable. It can be thought of as the initial design before the pattern is applied to it. Context refers to a recurring set of situations in which a pattern applies. The context of the patterns described in this paper is clear from the analysis model given in Figure 1 and therefore will not be included in the pattern description.
- Problem: a statement of the problem which describes its intent: the goals and objectives it wants to reach within the given context. Often the problem includes the applicability of the pattern. It could be phrased as a question.
- Solution: static relationships and dynamic rules describing how to realize the desired outcome, i.e., solve the problem. Sometimes possible variants or specializations of the solution are also described.
- Structure: describes the static structure of the pattern in terms of classes and their associations. It is usually shown using a UML class diagram.

4 THE CATALOG OF DESIGN PATTERNS

This section presents some patterns that capture solutions to problems that frequently occur (within structures, i.e. relations that exhibit the property of existential dependency and constraints of multiplicity) during software design. These patterns may specify a baseline for developing a design pattern language for detailed design.

The design model of every software application has to deal with associations among classes. An association is a structural relationship that specifies that objects of one class are connected to objects of another class. An association can be decorated with multiplicity, role names, and an association name. The multiplicity determines the minimum and maximum number of objects of one class that can be linked with one object of the other class. Each end of an association has one multiplicity. It has also a role



name (optional) which determines the role that a class plays in the association. During detailed design, an association may be refined into a dependency, an aggregation, or a composition relationship. In this section, the author will identify a few patterns related to the refinement of associations and their decorations such as multiplicity refinement. The quest for the patterns will result in discovering a few low level structural and behavioral patterns.

Structural patterns address issues concerned with the way in which classes are organized. To describe such patterns, object-oriented constructs and mechanisms such as inheritance, composition, aggregation, etc. are usually used. An example of such patterns is the Composite Pattern [Benn02]. Behavioral patterns, on the other hand, address the problems that arise when mapping responsibilities to operations in classes and when designing algorithms for the operation implementation. They usually describe how objects communicate to perform the system functionality. An example of such patterns is the State Pattern [Benn02].

Multiplicity Refiner

One-to-many relationships are very common in object-oriented models. In the model shown in Figure 1, many classes are associated with other classes with multiplicity 0 .. *. This means that zero or more objects of one class are associated with one object of the other class participating in the same relationship. For example, a professor may teach one or more course sections and no sections at all if he or she is on sabbatical. This multiplicity needs to be refined during the class design activity of RUP. This requires the introduction of a container class with operations to add and remove objects from the container. It might have operations to check the number of objects in the container, and maybe iterator operations. The Multiplicity Refiner pattern will help achieve this task.

Problem:

How to refine a one-to-many relationship between two classes during class design?

Solution:

The solution of this problem is obtained by introducing a container class to store the objects of the class that must be linked with the object of the other class. The container couples a participating object with an unlimited number of refined objects. The container class should add operations to retrieve the information needed from the container object.

Structure:

Figure 2 illustrates the structure of this pattern involving class A whose objects are associated with an unlimited number of objects of class B. The container class BList is added to store the objects of class B which are associated with one object of class A.

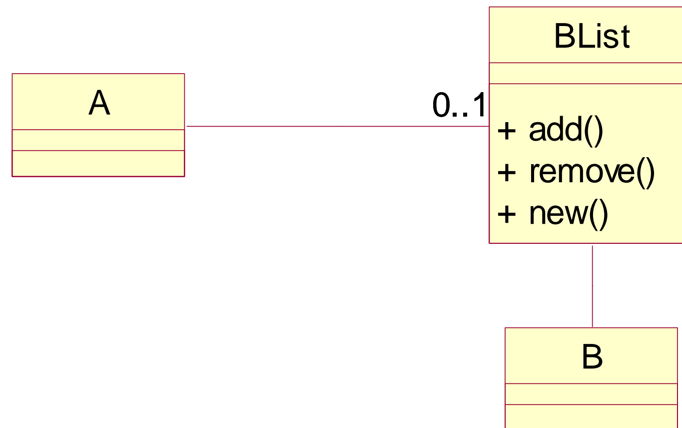


Figure 2: Structure of the Multiplicity Refiner Pattern.

Example:

Given the following portion of the model shown in Figure 1:

Applying the Multiplicity pattern produces the model in Figure 3:

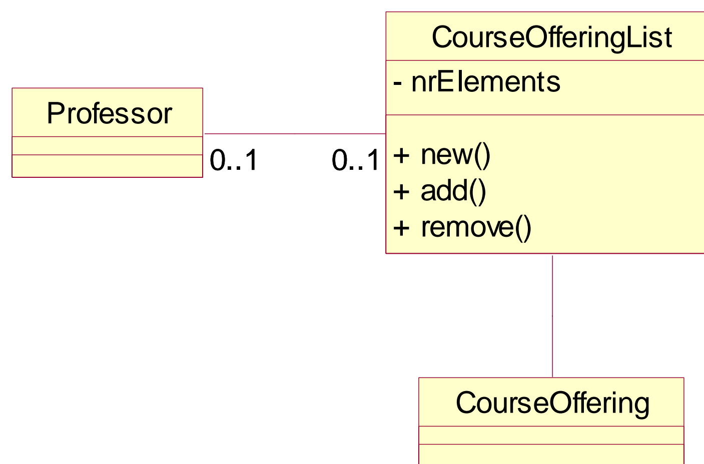
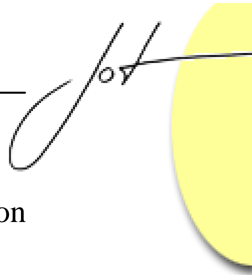


Figure 3: Application of the Multiplicity Refiner Pattern.

Genericity Refiner

Usually common container classes, such as List, are available from existing libraries. These classes can be used and should not be modeled. The container class of the



Multiplicity Refiner can instantiate a parameterized (generic) class if the implementation language provides generic container classes.

Problem:

How can a container class be mapped to a standard generic class in the implementation language during class design?

Solution:

Instantiate the standard container class using the refined class (whose objects must be contained) as the parameter.

Structure:

Figure 4 illustrates the structure of this pattern involving class List which is assumed to be a standard container class provided by the target language. The objects contained in the container class are objects from class B.

Example:

Given the model shown in Figure 3, the application of the Genericity Refiner pattern will produce the model in Figure 5.

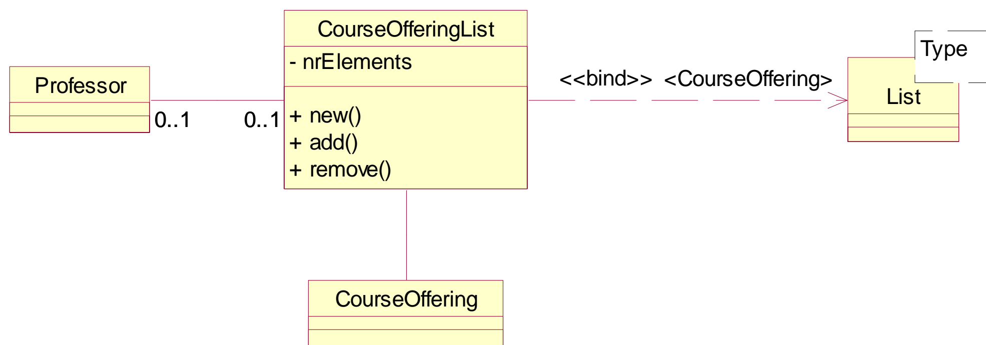


Figure 5: Application of the Genericity Refiner Pattern.

Optionality Refiner

In addition to showing the number of objects participating in a relationship, multiplicity answers the question whether or not the relationship is optional. This is usually indicated by using the multiplicity of 0..1.

Problem:

How can optional relationships given by the 0..1 multiplicity be refined during class design?

Solution:

Add the following operations to manipulate an optional relationship:

- addRefinedObject: to attach an object of the refined class to the object of the other participating class.
- removeRefinedObject: to detach the refined object from the other participating object.
- isRefinedObjectAssigned: to check if a refined object is attached to the other participating object.

Structure:

Figure 6 illustrates the structure of this pattern.

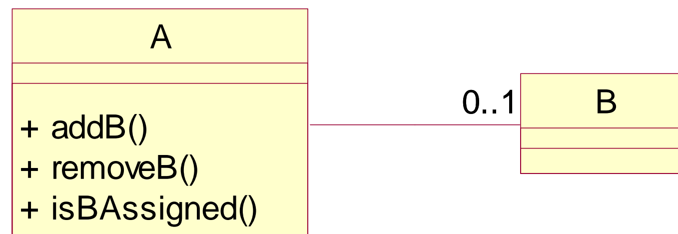


Figure 6: Structure of the Optionality Refiner Pattern.

Example:

Applying the Optionality Refiner pattern to the following association would produce the model in Figure 7.

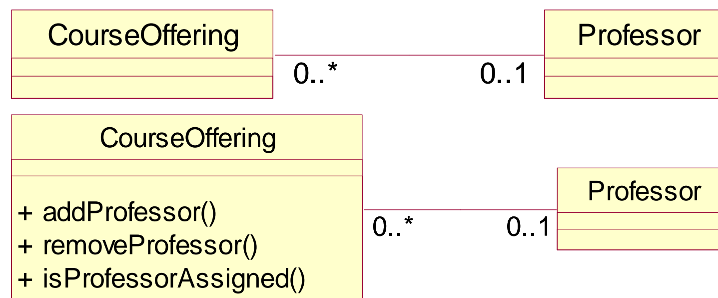
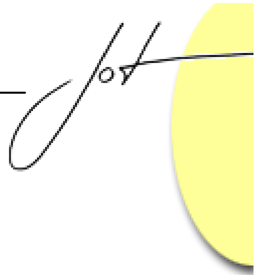


Figure 7: Application of the Optionality Refiner Pattern.

Association Class Refiner

When an attribute belongs to the association rather than to any of the two participating classes, an association class is added to include such an attribute. Figure 8 illustrates this by adding the association class PrimaryScheduleOfferingInfo to include the grade attribute.



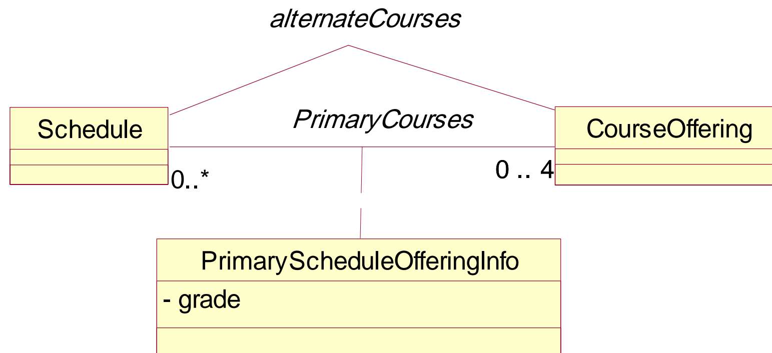


Figure 8: Association Class ScheduleOfferingInfo.

Problem:

How can an association class be refined during class design?

Solution:

Insert the association class between the two participating classes and switch multiplicities.

Structure:

Figure 9 illustrates the structure of this pattern involving the asscoiation class C with an attribute associationAttr which belongs to the association between class A and class B.

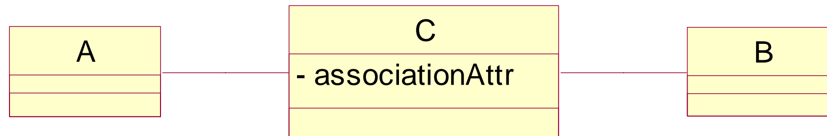


Figure 9: Structure of the Association Class Refiner Pattern.

Example:

Applying the Association Class Refiner pattern to the situation depicted in Figure 8 would produce the model in Figure 10.

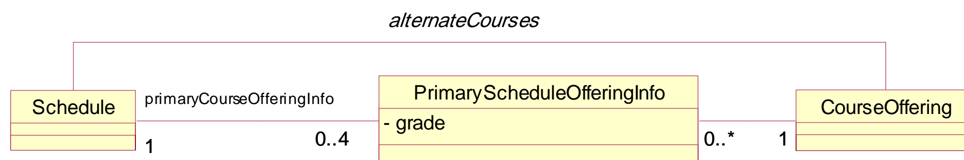


Figure 10: Application of the Association Class Refiner Pattern.

Dependency Selector

Dependency is a relationship between two model elements in which a change to one element (the supplier) may affect or supply information needed by the other element (the client) [Rumb05]. During analysis, many of the relationships between classes ar assumed to be structural relationships (associations). During detailed design, it should be decided what type of relationship is required; i.e. a structural or non-structural (dependency) relationship. A dependency relationship is the cheapest to maintain, easiest to utilize and benefits from the principle of encapsulation. Therefore, an association needs to be replaced by a dependency whenever possible. Usually, this step should be done after the define operations and methods step of class design in RUP, where the signature of each operation is fully specified and the algorithms for implementing the operations are determined. All non-field (local, global, parameter) visible associations should be replaced by dependency. One type of dependency relationships is addressed by the current pattern.

Problem:

When can an association be replaced by a dependency during class design?

Solution:

One case that is clear is between a container class and a class whose objects are supposed to be contained.

Structure:

Figure 11 illustrates the structure of this pattern.

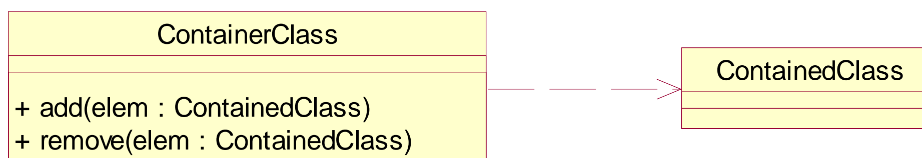
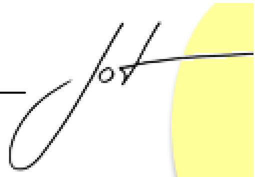


Figure 11: Structure of the Dependency Selector Pattern.

Example:

Applying the Dependency Refiner pattern to the situation depicted in Figure 5 would produce the model in Figure 12.



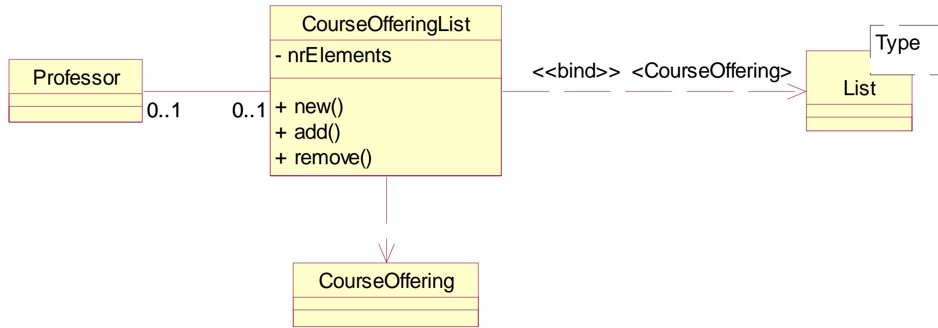


Figure 12: Application of the Dependency Selector Pattern.

Navigability Refiner

The navigability property on an association end indicates that it is possible to navigate from one class to the target class using the association. An arrowhead on the association end is used to show the navigable association direction. During analysis, the arrowheads are suppressed, meaning that associations are navigable in both directions. However, during class design, navigability should be reduced to one-way whenever it is possible because two-way associations are more expensive to implement. Representing a relationship between objects should be explicit, i.e., defining direction of navigation. It is usually not an easy task to determine the direction of navigability. However, it is clear in some cases such as a whole-part relationship, where the navigability is usually from the whole to the part. Another case is addressed by the current pattern.

Problem:

How can association navigability be determined during class design?

Solution:

One case is a one-to-many association, where the navigability is from the class representing the one to the class representing the many.

Structure:

Figure 13 illustrates the structure of this pattern.

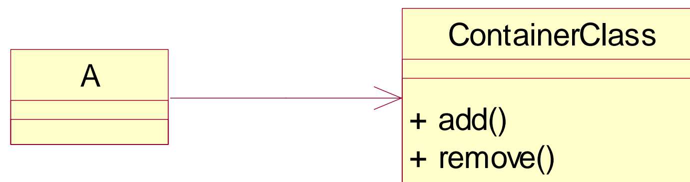


Figure 13: Structure of the Navigability Refiner pattern.

Example:

Applying the Navigability Refiner pattern to the situation depicted in Figure 3 would produce the model in Figure 14.

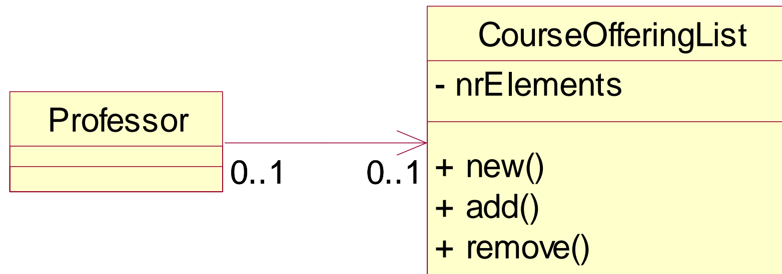


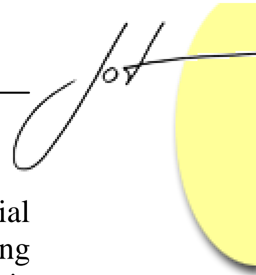
Figure 14: Application of the Navigability Refiner pattern.

5 TOOL SUPPORT

Having introduced a catalog of design patterns for detailed design activities in the previous section, the need for a CASE tool that will automate their usage, i.e., transformations is discussed in this section. Moreover, the key elements and requirements for such a tool will be identified. In fact, there is no lack of pattern catalogs in the literature that span all activities of software development: requirements specification, analysis, design, implementation, architecture, etc. There is however a lack of successful supporting CASE tools and methodologies that facilitate the construction of software from patterns.

Automated tool support for object-oriented methods is vital to the development of today's complex software systems. In fact, there is consensus on the importance and suefullness of tool support for patterns [Brag03, Budi96, Corn00, Flor97, Mape02]. Currently the pattern-based approach is done manually. Composing design patterns to develop application designs is a tedious task that involves integration of patterns. A tool support for development with patterns will eventually facilitate the analysis and design phases. It is believed that such automated tools will help designers concentarte on the big picture.

In fact, the majority of existing tools to support design patterns focus on support for high level patterns. For example, the tool proposed in [Yaco00] aims to integrate patterns at the architecture level. On the other hand, the tool proposed in this article aims to integrate design patterns such as the patterns described in this article at the detailed design level. One of the advantages of having tool support for detailed design activities is expedite the design phase. Even though such a tool can never replace a human expert, it can help him to identify types of design patterns that can be used to solve a particular design problem. The tool can also help to compare design alternatives by evaluating each alternative in terms of quality factors and by evaluating the results.



The ultimate objective is to develop an approach that scales up to large industrial software systems. Our aim is to improve the software development process by focusing on the cyclic and evolutionary nature of the software life cycle. The focus of this paper is on developing patterns of relationships between classes and objects that define preferred solutions to common object-oriented design problems. The next step, work already started, will be the development of a prototype of a tool (wizard) that will take an analysis model and transform it to a detailed design model. Such a tool will allow a designer to select an association from an analysis model and then apply any of the described patterns. The tool will automatically replace the classes with the necessary refined classes and their operations.

An advantage of using the patterns introduced in the previous section is that they make the design model close to the implementation model, thereby facilitating optimal code generation for a target language. Note that no new language constructs are needed to implement these patterns in the target language. A disadvantage of implementing these patterns, on the other hand, is that the design model will become bigger and therefore more complex. This necessitates the use of a tool to introduce the patterns and manage the model. Moreover, the tool will allow the designer the option of viewing the model with or without these low level patterns.

The tool being developed is meant to be integrated with an object-oriented software modeling tool to fully benefit from the patterns discussed in this article. This can be achieved by extending an existing visual modeling tool such as IBM Rational Rose with the functionality provided by this tool. In this case, the work to be done to include support for automatic implementation of these patterns would be minimal.

6 CONCLUSIONS AND FUTURE WORK

This paper has introduced a catalog of patterns for detailed design. The catalog consists of six related patterns that recur in every application. This pattern catalog can form the basis of a pattern language for detailed design. The pattern catalog is not complete in the sense that it can be extended with other patterns for generalization/specialization relationship as well as the refinement of association into dependency, aggregation, and composition relationships. It is believed that such a pattern language will help create detailed design models that are close to the implementation language, which will facilitate the generation of code from the design model easily and more efficiently. A CASE tool is being designed to automate the transformation of an analysis (initial design) model to a detailed design model using the proposed pattern catalog. Such a tool will improve the productivity of the developers and shorten the time needed to complete the design.

REFERENCES

[Benn02] Bennett S. et al, "Object-Oriented Systems Analysis and Design Using UML", McGrawHill, 2002.

[Brag03] Brag R. and Masiero P, "Building a Wisard for Framework Instantiation Based on a Pattern Language", *Proceedings of OOIS 2003*, 2003.

[Budi96] Budinsky F. et al, "Automated code generation from design patterns", *IBM Systems Journal*, 1996.

[Busc01] Buschman F. et al, "Pattern-Oriented Software Architecture", Wiley, 2001.

[Corn00] Cornils A. and Hendin G., "Tool Support for Design Patterns based on Reference Attribute Grammers", *Proceedings of WGA00, 2000*.

[Flor97] Florijn G. et al, "Tool support for object-oriented patterns", *Proceedings of ECOOP'97*, 1997.

[Fowl97] Fowler M., "Analysis Patterns: Reusable Object Models", Addison-Wesley, 1997.

[Gamm95] Gamma E. et al, "Design Patterns: Elements of Reusable Object-Oriented software", Addison-Wesley, 1995.

[Larm02] Larman C., "Applying UML and Patterns", Prentice Hall, 2002.

[Lee03] Lee T. et al, "Object-Oriented Design of RTI Using Design Patterns", *Proceedings of OOIS2003*, 2003.

[Mape02] Mapelsden D. et al, "Design Pattern Modeling and Instantiation using DPML", *Proceedings of Tools Pacific'02*, 2002.

[Quat00] Quatrani T., "Visual Modeling with Rational Rose 2000 and UML", Addison-Wesley, 2000.

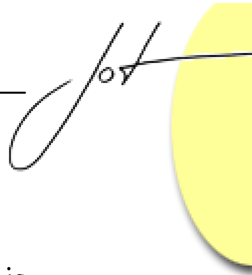
[Rati00] Rational University, "Object-Oriented Analysis and Design Using UML, Student Manual", Version 2000, Part# 800-011765-002, 2000.

[Rumb05] Rumbaugh J. et al, "The Unified Modeling Language Reference Manual", Second Edition, Addison-Wesley, 2005.

[Shal02] Shalloway A. and Trott J., "Design Patterns Explained", Addison-Wesley, 2002.

[Yaco00] Yacoub S., Xue H., and Ammar H. "Automating the Development of Pattern-Oriented Designs for Application Specific Software Systems", *Proceedings of ASSET'00,* 2000.

[Yaco04] Yacoub S. and Ammar H., "Pattern-Oriented Analysis and Design", Addison-Wesley, 2004.



About the author

Walid Al-Ahmad is an associate professor of Computer Science. He is currently working at New York Institute of Technology - Bahrain Campus. He is the coordinator for the School of Engineering and Technology. His research centers around improving support for object-oriented concepts and principles in OOPLs as well as new emerging software approaches such as components, design patterns and web services. He can be reached at w.alahmad@nyit.edu.bh