UML Support for Designing Software Systems as a Composition of Design Patterns

Sherif M. Yacoub[1] and Hany H. Ammar[2]

[1] Hewlett-Packard Labs, 1501 Page Mill, MS 1L-15,
Palo Alto, CA 94304, USA
sherif_yacoub@hp.com
[2] Computer Science and Electrical Engineering Department, West Virginia University
Morgantown, WV 26506, USA
hammar@wvu.edu

Abstract. Much of the research work on design patterns has primarily focused on discovering and documenting patterns. Design patterns promise early reuse benefits at the design stage. To reap the benefits of deploying these proven design solutions, we need to develop techniques to construct applications using patterns. These techniques should define a composition mechanism by which patterns can be integrated and deployed in the design of software applications. Versatile design models should be used to model the patterns themselves as well as their composition. In this paper, we describe an approach called *Pattern-Oriented Analysis and Design* (POAD) that utilizes UML modeling capabilities to compose design patterns at various levels of abstractions. In POAD, the internal details of the pattern structure are hidden at high design levels (pattern views) and are revealed at lower design levels (class views). We define three hierarchical traceable logical views based on UML models for developing pattern-oriented designs; namely the *Pattern-Level* view, the *Pattern Interfaces* view, and the *Detailed Pattern-Level* view. The discussion is illustrated by a case study of building a framework for feedback control systems.

Keywords: Pattern-Oriented Design, Design Patterns, and Pattern Composition.

1 Introduction

Patterns are reusable good-quality design practices that have proven useful in the design of software applications [2,11]. Patterns can help in leveraging reuse to the design level because they provide a common vocabulary of designs and they are proven design units from which more complex applications can be built. Much work has focused on documenting patterns [e.g. 2,11,12,16]. Other work is concerned with applying these reusable designs in constructing applications [e.g. 3,10,14,15]. We can generally classify design approaches that utilize patterns as:

1. Adhoc. A design pattern records a solution and forces and consequences of applying this solution. However, this is not usually sufficient to systematically develop applications using patterns. For instance, the coincidental use of a *Strategy* pattern [2] in the implementation of a control application is not a systematic approach to deploy patterns. This is simply because there is no process to guide the development and to integrate the pattern with other design artifacts.

2. Systematic. A systematic approach to design with patterns goes further beyond just applying a certain pattern. Systematic approaches can be classified as:

a) Pattern Languages. A pattern language provides a set of patterns that solve problems in a specific domain. Pattern languages not only document the patterns themselves but also the relationships between these patterns. They imply the process to apply the language to completely solve a specific set of design problems.

b) Development processes. A systematic development process defines a pattern composition approach, analysis and design steps, design models, and tools to automate the development steps. Such development process produces consistent designs each time the process steps are conducted.

We are concerned here with systematic development processes because they are the way to repeatable software design practice. To improve the practice of *systematically* deploying design patterns, we need to define methodologies to construct applications using patterns and support these methodologies with appropriate modeling languages. In this paper, we discuss a process to develop pattern-oriented applications using UML modeling capabilities. Specifically, we discuss using UML in the Pattern-Oriented Analysis and Design (POAD) process [10,17,31]. POAD uses design patterns as building blocks. The design of an application is built by gluing together these construction fragments and defining dependencies and collaboration between participating patterns. To make this approach availing, we need to define modeling artifacts that support its automation. Applications developed using this approach are object-oriented in nature. Thus, the Unified Modeling Language [1,13] is used in each step.

In this paper, we discuss UML support for modeling design patterns and developing pattern-oriented designs. We show how to use UML modeling capabilities and the POAD process to develop logical design views that capture relationship between patterns while hiding details not utilized directly in the design. We then show how to use these views to overlap participants of patterns to produce a denser and a more profound class diagram. To illustrate the application of the proposed models and process, we use a case study of building a framework for feedback control systems.

2 Stringing Versus Overlapping Patterns

"*It is possible to make buildings by stringing together patterns in a rather loose way. A building made like this, is an assembly of patterns. It is not dense. It is not profound. But it is also possible to put patterns together in such a way that many patterns overlap*

in the same physical space: the building is very dense; it has many meanings captured in small space; and through this density, it becomes profound."[7]

In the field of civil engineering, Alexander *et.al.* discuss techniques for composing patterns as they experienced in making buildings. They compare two approaches: stringing and overlapping patterns. Many of these principles apply to the design of software systems as well. Inspired by Alexander's approaches to make buildings, consider the two approaches to build software applications using design patterns:

1) *Stringing patterns.* In this design approach, patterns are glued together to compose an application design. The glue here could simply be UML relationships between patterns as packages (for example dependency between packages) or UML relationships between participants of the patterns (for example UML association, dependency, etc. between classes of one pattern and classes of another pattern). The design is a loose assembly of patterns because it is made by simply stringing patterns and using all the internal participants of a pattern as independent design constructs. The design is neither dense nor profound. It is not dense because we end up with a design that has a large population of classes. It is not profound because many classes have trivial responsibilities. The reason is that the design of many of these patterns has several classes that are only responsible for forwarding to other classes, acting as an interface to the internal design of the pattern, or representing a class that is intended to be part of the external system design not the internal design of the pattern (i.e. a client class of a pattern).

2) *Overlapping patterns.* This approach advocates that many patterns should overlap in the same logical design. Overlapping means that a class, as a participant in one pattern, could be at the same time a participant of another pattern in the same application design. For instance, consider gluing together the *Strategy* and the *Observer* patterns [2]. Overlapping these two patterns could mean that the abstract `Strategy` class of the *Strategy* pattern plays the role of abstract `Subject` class in the *Observer* pattern. The designer will use one application class to play the role of both participants. As a result, the design is dense and it becomes more profound. It is dense because we end up having fewer classes in the application design than the total number of classes in the patterns used to develop that design. It is profound because each class carries out several responsibilities.

With the overlapping patterns approach, we gain the advantage of having less number of classes in the application design than the one produced by stringing patterns. However, there is one salient disadvantage. The pattern boundary is lost and patterns become hard to trace. With stringing patterns, we can always identify the pattern by circling the classes that implement it. When circling the classes of a pattern in the overlapping pattern design, we end up with so many intersecting circles.

As an example, consider an application in which the designer has decided to use the *Reactor* pattern [8] and the *Composite* pattern [2]. We will use these two patterns in the sequel to illustrate the difference between the overlapping and stringing approaches. The class diagram model for each of the two patterns is shown in Figure 1.

The *Reactor* pattern is a robust design for a system that receives events, manages a set of event handlers, and dispatches the event to the appropriate handler. It consists of: the abstract `EventHandler` class which is the interface that all the concrete

event handlers have to comply with; the `Reactor` class which is the class responsible for scheduling events and dispatching them to event handlers according to the type of the event; and finally the `ConcreteEventHandler` classes which implement the `EventHandler` interface.

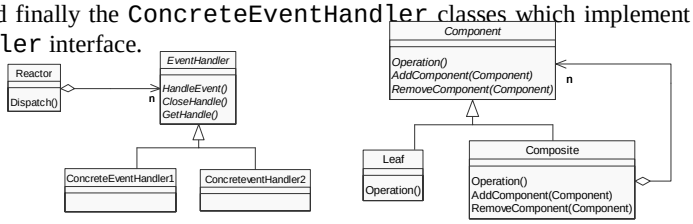

Fig. 1 The class diagram for a) *Reactor* pattern, and b) *Composite* pattern

The *Composite* pattern is a robust design for a system that provides a unique interface to a complex structure of objects that could be simple or composites. It is composed of: the `Component` class which is the interface for the structure; the `Leaf` class which implements the `Component` interface but does not contain other objects of type `Component`; and finally the `Composite` class that implements the `Component` interface and consists of other components that it manages.

Consider the case where we want to use these two patterns in designing a reactive system. When using a *Reactor* pattern, we might find that the handlers for the application specific events are not simple objects; instead, they could be complex objects containing other objects that react as well to the events. Hence, we decide to use a *Composite* pattern for the handlers. Now, how do we glue these two pattern?

The first solution is to string the two patterns together by establishing a relationship between the `Component` class of the *Composite* pattern and the `EventHandler` class of the *Reactor* pattern. By stringing the two patterns, we develop the design shown in Figure 2, which contains all the classes of the two patterns.

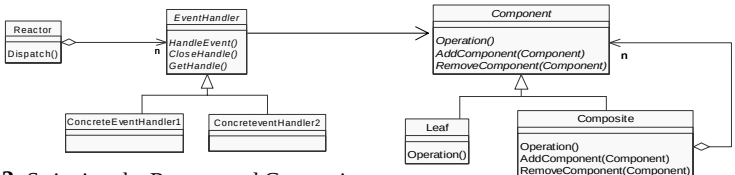

Fig. 2. Stringing the Reactor and Composite patterns

The design in Figure 2 is not profound because it assumes that the handlers use or reference composite components while in reality the handlers *are* the composite components.

The second solution is to overlap the two patterns. We overlap the `EventHandler` of the *Reactor* pattern and the `Component` class of the *Composite* pattern and both roles are integrated in one class call it `EventHandlerComponent`. This class will have the methods from both classes. Consequently, the concrete event handlers become concrete classes derived from the `EventHandlerComponent` class. The design of the overlapped pattern design is shown in Figure 3

The question that rises here is: are these two approaches independent? Must we construct a design that is either a sparse assembly or a condensed overlap of patterns? Can we use both?

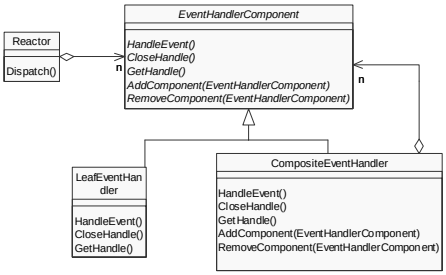

Fig. 3. Overlapping the *Reactor* and *Composite* patterns

Clearly, the first approach, assembling and stringing patterns, is avoided by many designers. This can be attributed to the perceivable disadvantages of simply assembling patterns to produce designs. It is, however, an easy approach to practice. The stringing pattern approach provides good traceability from high-level designs, in terms of patterns, to lower-level designs, in terms of classes. We can simply encapsulate the classes of a pattern in one package or a template package [5], which will become the high level view and use the pattern classes in the class diagram model which will become the low level design.

The Pattern-Oriented Analysis and Design (POAD) approach reaps benefits from both worlds; the stringing and overlapping patterns worlds. It makes use of the simplicity and traceability of the stringing-patterns approach and the density and profoundness of the overlapping-patterns approach. In POAD, the two approaches are not independent and in fact they could be integrated in one process. POAD starts by assembling patterns at a higher level of abstraction using the stringing approach, provides models to trace the patterns to lower levels of abstraction, and then allows the designer to integrate lower level classes to produce dense and profound designs.

3 Pattern-Oriented Analysis and Design with UML

In this section, we discuss a pattern oriented analysis and design process that utilizes UML modeling capabilities at various development steps. The following subsections describe each step with application to the development of a feedback control framework. Feedback systems are commonly modeled using block diagrams. The design framework that we develop in this paper is based on design patterns as building constructs. The framework is documented at various design levels using UML models and is reusable as an initial phase in designing feedback control applications.

3.1 Analysis

The purpose of this step is to analyze the application requirements and decide on the set of design patterns that will be used in designing the system. To design a feedback control system, the specification and description of the system configuration and its components must be put into a form amenable for analysis and design. Three basic representations (models) of components and systems are used extensively in the study of control systems: mathematical models, block diagrams, and signal-flow graphs.

Referring to control literature [e.g. 9], the generic block diagram of feedback systems represents an initial architecture documentation to start with. Figure 4 illustrates the block diagram that is often used to describe a feedback control system.

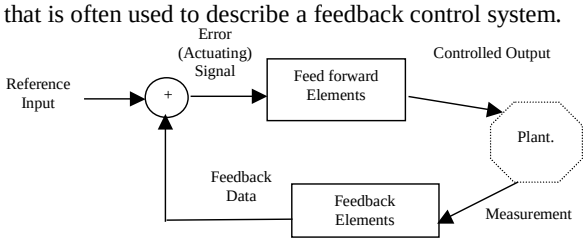

Fig. 4. Block diagram for a feedback control system

The portion of a system to be controlled is usually called the *Plant*. An output variable is adjusted as required by the error signal. This error signal is the difference between the system response as measured by the feedback element and the reference signal, which represent the desired system response. Generally, a controller is required to process the error signal such that a certain control strategy will be applied. Using the generic block diagram of a closed loop control system, the system is decomposed into: a *feedforward* component that processes the error data and applies a control algorithm to the plant; a *feedback* component that measures data from the plant, processes it, and provides the feedback data; an *error calculation* component that compares the input and feedback data and produces the error; and the *plant* that is an external component on which control is applied and from which measurements are taken.

3.2 Pattern Selection

We analyze the responsibilities and the functionalities of each component and identify candidate patterns that could provide a design solution for each component. In doing so, we have considered the design problem that we want to solve and match it to the solution provided by general purpose design patterns [e.g. 2, 11, 12]:

1. The *feedforward* component implements some sort of a control strategy. The change in the control strategy should be flexible and hidden from any calls and invocations form any other component. For example, the *feedforward* component should provide the same interface to the rest of the components in the system while the framework can provide the flexibility to plug in and plug out different control strategies. If we consider this as the design problem that we want to solve and search for patterns whose intent is to solve similar problems, we find that a *Strategy* pattern [2, pp315] is a good candidate for this task.
2. The *feedback* component receives measurements and applies a feedback control strategy. It feeds the result to the error calculation component. The measurement unit observes and measures data from the plant and feeds it to the feedback branch. Thus, measurement observations can be communicated to the feedback controller using the *Observer* pattern [2, pp293]. Thus we can use the *Observer* pattern to loosen the dependency between the objects doing the plant observation and those actually doing the feedback control. The measured data is fed to the feedback control strategy, which - similar to the *feedforward* component- should

provide flexibility to plug in and plug out different feedback control strategies. This can be implemented using another *Strategy* pattern [2, pp315].

3. In the *error calculation* component, the feedback controller notifies the error calculation unit with the feedback data. The feedback controller can be viewed as the subject that notifies the error calculator with changes in the feedback data. Error calculation is done whenever feedback data becomes available, at that moment, this data is compared with the persistent input data. Thus, an *Observer* pattern [2, pp293] can implement this behavior.
4. If we examine the data manipulated in the feedback system, we find that the system handles: measurement data that is measured from the plant; feedback data that is the result of processing the measured data by the feedback element; and finally the error data that is the result of processing the feedback data and the input data. Data of different types need to be exchanged between the framework components. We can use a *Blackboard* pattern (a modified version of the blackboard patterns in [24, 11]) for managing the system repository.

In choosing these patterns, we consider how the pattern solves the design problem and the intent of the pattern. In summary, a *Strategy* pattern is selected for the *feedforward* component, an *Observer* and a *Strategy* pattern are selected for the *feedback* component, an *Observer* pattern is selected for the *error calculation* component, and a *Blackboard* pattern is selected as the system repository. In this small example, it was obvious which patterns could be used. In other complex example, the analyst could use UML use cases and sequence diagrams to understand the functionality required by each component.

3.3 Constructing *Pattern-Level* Diagrams

In this step, we create instances of the selected patterns and identify the relationships between these instances. As a result, a *Pattern-Level* diagram of the system is developed.

First, we create pattern instances. In the previous step, we have selected to use two *Strategy* patterns one in the *feedforward* component and the other in the *feedback* component. Thus, we use the instances *FeedforwardStrategy* and *FeedbackStrategy* of type *Strategy* pattern in the design of the *feedforward* and *feedback* components respectively. We have also selected to use two *Observer* patterns one for the *feedback* component and the other for the *error calculation* component. Thus, we use a *FeedbackObserver* instance of type *Observer* pattern to observe and measure data from the plant and an *ErrorObserver* instance of type *Observer* pattern to calculate the error. We use a *Blackboard* of type *Blackboard* pattern to manage the system data repository. This is just giving domain specific names to abstract patterns types (templates).

Second, we define dependency relationships between pattern instances. The *FeedbackObserver* uses the *FeedbackStrategy* to apply a feedback control algorithm, which in-turn, uses the *ErrorObserver* to calculate the error. The *ErrorObserver* uses the *FeedforwardStrategy* to apply a forward control algorithm. The *Blackboard* is used by all patterns to store and retrieve data.

Finally, we use the pattern instances and their relationships to construct the *Pattern-Level* diagram as shown in Figure 5. We use UML stereotypes to show the type of the pattern instance.

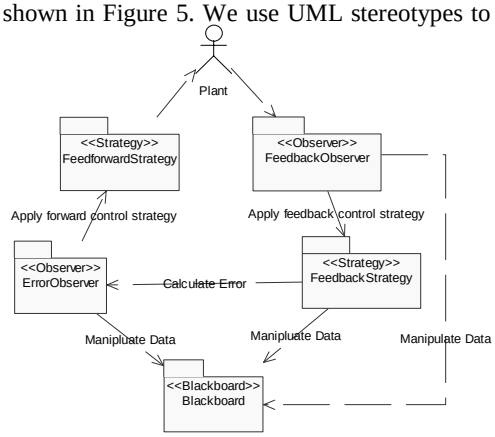

Fig. 5. A *Pattern-Level* diagram for feedback control systems

The product of this process is the *Pattern-Level* diagram of the framework. It describes the architecture of a feedback system using design patterns, which explains why the names "*Pattern-Oriented Analysis and Design*" is used. During the design or design refinement phases we could discover that a selected pattern has limitations or impacts on other design aspects. In this case, the designer would revisit this design level to choose another pattern, replace previous choices, or create a new pattern dependency or a new uses relationship.

3.4 Constructing *Pattern-Level with Interfaces* Diagram

In this step, the dependency relationship between patterns in the *Pattern-Level* view is further traced to lower level design relationships between pattern interfaces.

First, we declare interfaces for the patterns used in each *Pattern-Level* diagram (only one diagram for the feedback system). The *Strategy* pattern has the class `Context` as the interface to the encapsulated control strategy. The *Observer* has two interfaces that allow coordinating the subject observed with its observer. These interfaces are implemented by the `notify()` interface in the subject and the `update()` interface in the observer. The *Blackboard* pattern has the interfaces to get and store data in the repository, these interfaces are implemented by the `getData()` and `setData()` methods. Then, we identify the relationship between pattern interfaces by translating all dependency relationships between patterns in a *Pattern-Level* diagram to relationships between interface classes and/or interface operations. The product of this process is the *Pattern-Level with Interfaces* diagram. Figure 6 illustrates the *Pattern-Level with Interface* diagram for the feedback control framework.

As an example consider the relationship between the *FeedbackObserver* and the *FeedbackStrategy* pattern instances in the *Pattern-Level* view. The relationship between these two patterns at the *Pattern-Level* view is that the *FeedbackObserver* uses the *FeedbackStrategy* to apply a feedback control strategy whenever the measurement data is ready. The interfaces of the *FeedbackObserver* are the `Update()` and the

`notify()` interface methods. The interface of the *FeedbackStrategy* is the `Context` interface class. Thus the relationship between these two patterns is translated to a relationship between the `Update()` interface of the earlier and the `Context` interface of the latter. Similarly, all pattern relationships of Figure 5 are translated to relationships between the pattern interfaces in Figure 6.

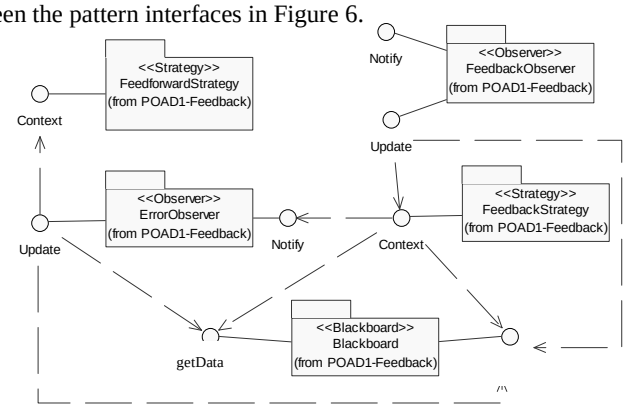

Fig. 6. A *Pattern-Level with Interfaces* diagram for feedback control systems

3.5 Constructing *Detailed Pattern-Level* Diagrams

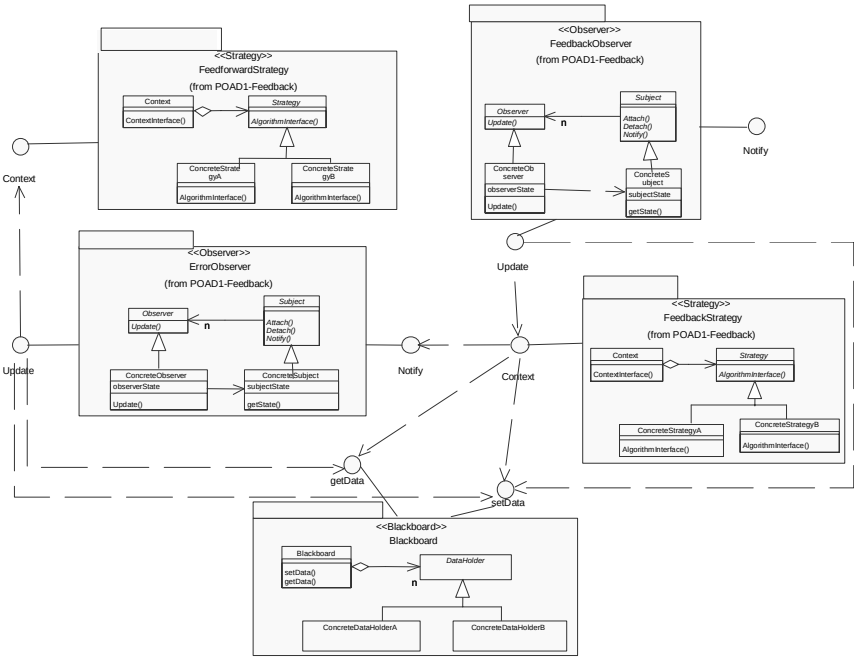

Fig. 7. A *Detailed Pattern-Level* diagram for feedback control systems

To construct the *Detailed Pattern-Level* diagram, we express the internals (i.e. participants) of each instantiated pattern in the *Pattern-Level with Interfaces* diagram. Since we have used pervasive design patterns in developing the feedback control framework, their structure can be found in the literature. For example, the class diagram model for the *Strategy* and *Observer* patterns is documented in [2]. Figure 7 illustrates the *Detailed Pattern-Level* diagram for the feedback pattern-oriented framework. Note that we do not take any additional design decisions in this step. With the appropriate tool support Figure 7 is a direct generation from the *Pattern-Level with Interfaces* diagram by simply retrieving the class diagram model from a pattern database.

3.6 Instantiating Pattern Internals

In this step, we add domain specific nature to the *Detailed Pattern-Level* diagrams by renaming internal pattern classes according to the application domain, choosing names for pattern participants that are meaningful in the application context, and defining domain specific names for operations in the patterns. Due to space limitation, we will illustrate few examples only in the sequel.

Instantiating the *ErrorObserver* Pattern

The error calculation component consists of the *ErrorObserver* pattern, which is composed of:

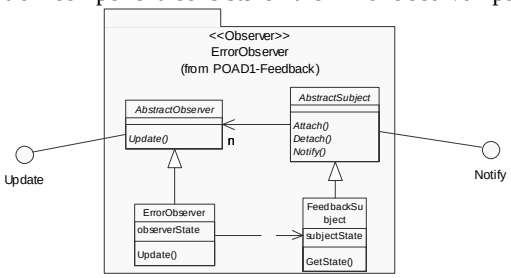

Fig. 8. Instantiating the *ErrorObserver* pattern

- `AbstractObserver.` An updating interface for objects that are notified of changes in the subject.
- `AbstractSubject.` An interface for attaching and detaching observers. It knows about its observers that ought to be notified of a subject's change.
- `ErrorObserver.` It is a concrete observer that maintains a reference to the `FeedbackSubject`, reads the feedback data after being processed by the feedback strategy, analyzes the feedback data with respect to the reference input data, and stores the error in the blackboard. It implements `AbstractObserver` update interface.
- `FeedbackSubject`. It is a concrete subject that sends notification to the concrete observers of new data received from the feedback component.

Instantiating the *FeedbackObserver* Pattern

The *FeedbackObserver* is used in the feedback component and is composed of:

- `AbstractObserver` and `AbstractSubject`. They play an interface role similar to that of the `ErrorObserver` pattern.

- `MeasurementSubject.` It receives measurement notifications from the plant and notifies its observer `FeedbackObserver` that a measurement is ready.
- `FeedbackObserver.` When notified by changes in the plant (through the `MeasurementSubject`), it pulls the data identifier from the subject (using the pull mode of the *Observer* pattern) and invokes the feedback controller to process the measured data.

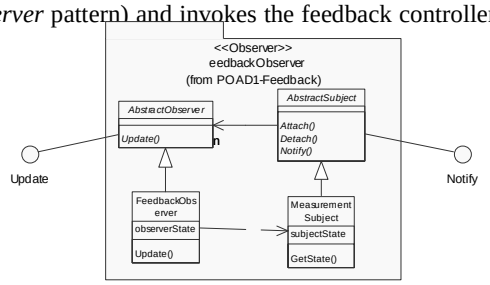

Fig. 9. Instantiating the *FeedbackObserver* pattern

Instantiating the *FeedbackStrategy* Pattern

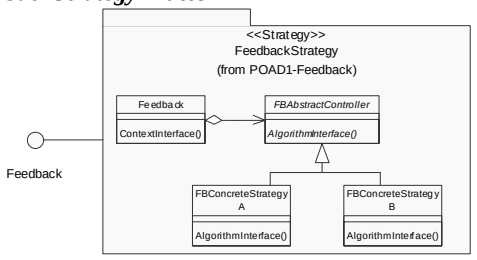

Fig. 10. Instantiating the *FeedbackStrategy* pattern

The *FeedbackStrategy* pattern is composed of:

- `Feedback.` It is the context of the feedback control strategy. It is configured with a feedback control strategy object through a reference to an `FBAbstractController.`
- `FBAbstractController`: It is the interface for all feedback control strategies. The `Feedback` uses this interface to call the feedback concrete algorithm.
- `FBControlStrategyA`, and `FBControlStrategyB.` They represent concrete implementations for feedback control strategies.

The `FeedbackObserver` invokes the control routine of the `Feedback` that applies the feedback control strategy required from the component. The `Feedback` class interacts with the `FeedbackSubject` of the observer pattern in the error calculation component and invokes its `notify()` procedure. This establishes the link between the feedback component and the error calculation component.

Two features can help the designer keep track of the patterns. First, the three models *Pattern-Level* diagram, the *Pattern-Level with Interfaces* diagram, and the *Detailed Pattern-Level* diagram provide a documentation of the framework as a composition of patterns. Second, with the appropriate tool support, the *renaming* process is not an *editing* process. In editing we simply change the names and the old names are lost. But in the renaming process of a class, the tool support for POAD should provide a

system with memory to keep the history of the changed name specifically in pattern instantiation.

3.7 Developing an Initial Class Diagram

From the *Detailed Pattern-Level* diagram, we use pattern interfaces and the instantiated details of pattern internals to construct a UML class diagram. The class diagram that is developed at this phase is an initial step to develop the static design model of the pattern-oriented framework. Figure 11 illustrates the class diagram for the framework. It can be recognized that the patterns are still notable in the class diagram as shown by the dotted boxes around the classes. As part of POAD, all the models in Figure 6 through Figure 11 are saved as analysis and design models. It is the role of a tool support to preserve these models and provide the necessary traceability.

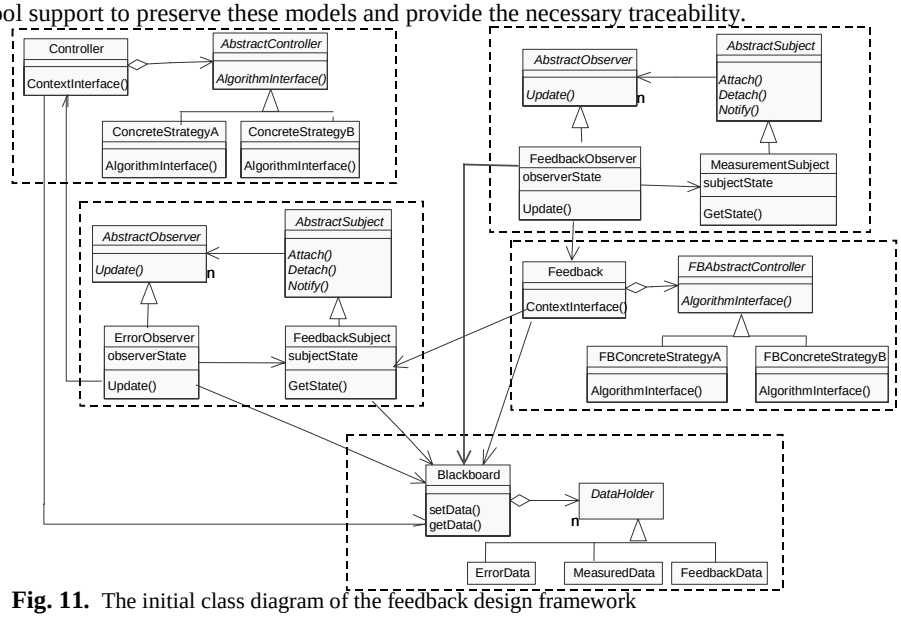

Fig. 11. The initial class diagram of the feedback design framework

The class diagram obtained from gluing patterns together at the high-level design is neither dense nor profound because we just stringed the patterns together. It has many replicated abstract classes due to the fact that we used multiple instances of the same pattern. For example we used the *FeedbackStrategy* and the *FeedforwardStrategy* instances of type *Strategy* pattern. It also has many classes with trivial responsibilities because many classes are just doing forwarding of messages to internal participants of the pattern. In the following step we use reduction and grouping mechanisms to optimize the UML design diagrams obtained initially in the previous step.

3.8 Design Refinement

The complexity of the framework can be reduced by eliminating replicated abstract classes. A pattern has one or more abstract classes. Since the same pattern type is

used in more than one instance, we expect to find similar abstract classes. For example, the *Observer* pattern is used in the feedback component and in the error component. The classes `AbstractObserver` and `AbstractSubject` are replicated. Similarly, the abstract class `AbstractController` of the strategy pattern used in the *feedforward* and *feedback* components. Therefore, the replicated classes are eliminated and only one common version of the abstract classes is used. This step is not usually applicable to all designs because the interfaces offered by abstract classes may substantially differ and hence we might not be able to merge these two abstract classes. For the feedback control system this was possible. In general, this is an activity that the designer might consider doing as part of the development process.

More optimization in class usage can be achieved by merging concrete classes together depending on their interaction and responsibilities. This step mainly depends on the framework designer skills. From Figure 11, we find that the classes `FeedbackObserver`, `FeedbackSubject` and `Feedback` perform highly related functions, which are summarized as receiving measurement notification, applying control strategy, and notifying the error component that the feedback data is ready. Instead of implementing a primitive function in each class, the three classes are merged into one class `FeedbackSubjectObserver`, which carries out the responsibilities of the three classes. Figure 12 illustrates the refined class diagram of the framework

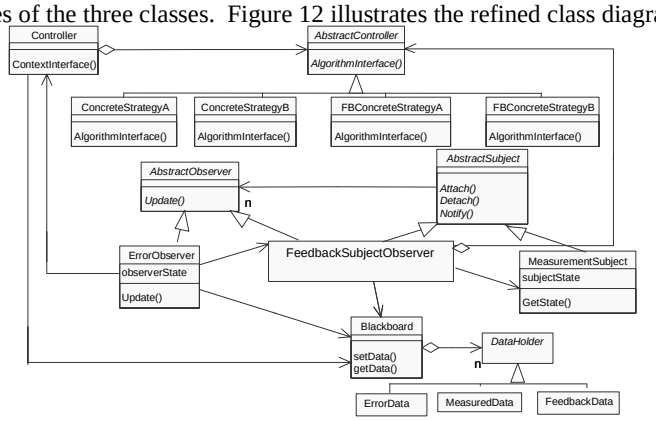

Fig. 12. The refined class diagram for the feedback design framework

It could become difficult to identify the patterns at this level because at this level we are using domain specific classes. This problem has always been recognized in many techniques that directly use patterns at the class diagram level without developing higher level design models: patterns are lost and are not traceacble [25,26]. POAD has one particular advantage. When applying POAD, we keep all the models developed through out the development lifecycle. These models are traceable bottom-up from the class level to the pattern level and top-down from the pattern level to the class level.

As an example of top-down traceability, we can identify the pattern participants in the above class diagram. For example, *FeedforwardStrategy:Strategy*, is composed of the classes `Controller`, `AbstractController`, `ControlStrategyA`, `Con-`

trolStrategyB. Another example is *ErrorObserver:Observer*, which is composed of the classes AbstractSubject, AbstractObser-ver, FeedbackSubjectObserver (a concrete subject) , and ErrorObserver (a concrete observer). As an example of bottom-up traceability, we find that the class FeedbackSubjectObserver is a common participant in multiple patterns. It is the observer for MeasurementSubject in the FeedbackObserver. It acts as a controller in the FeedbackStrategy that invokes a concrete control strategy to be applied on the FeedbackData. It acts as a subject for the ErrorObserver.

4 Related Work

Several successful experiences have reported on the advantages of using patterns in designing applications [15, 27]. These experiences do not follow a systematic method to develop applications using patterns. Systematic development using patterns utilizes a composition mechanism to glue patterns together at the design level.

Generally, we categorize composition mechanisms as behavioral and structural compositions. Behavioral composition approaches are concerned with objects as elements that play several roles in various patterns. These approaches are also known in the OO literature as interaction-oriented or responsibility-driven design [28]. Reenskaug [29,30] developed the Object Oriented Role Analysis and Software Synthesis method (OORASS, later called OOram). The method uses a role model that abstracts the traditional object model. Riehle [18] uses role diagrams for pattern composition. Riehle focuses mainly on developing composite patterns, which are compositions of patterns whose integration shows a synergy that makes the composition more than just the sum of its parts. The approach by Jan Bosch [26] uses design patterns and frameworks as architectural fragments. Each fragment is composed of roles and components that are merged with other roles to produce application designs. Lauder et. al. [20] take a visual specification approach to design patterns. They utilize constraint diagrams that are developed by Kent [19] together with UML diagrams.

Structural composition approaches build a design by gluing pattern structures that are modeled as class diagrams. Structural composition focuses more on the actual realization of the design rather than abstractions as role models. Behavioral composition techniques such as roles [30,18] leave several choices to the designer with less guidelines on how to continue to the class design phase. Keller and Schauer [21,22] address the problem of software composition at the design level using design components. Their approach and ours share the same objective of creating software designs that are based on well-defined and proven design patterns packaged into tangible, customizable, and composable design components. Larsen [23] takes a structural approach to glue patterns by mapping the participants of a pattern into implementation components. POAD shares the same concept of defining interfaces for patterns.

Xavier Castellani and Stephan Y. Liao [4] propose an application development process that focuses on the reuse of object-oriented application design. D'Souze et. al. [5,6] define a component-based approach to develop software that is heavily based on interfaces at both the design and implementation level. D'Souza's approach is general

in addressing software development issues such as composing physical components, distribution of components, and business driven solutions, etc. Modeling patterns as template packages is similar to the pattern level view developed in earlier development models used in POAD.

5 Conclusion and Future Work

The work in this paper stems from the need to develop systematic approaches to glue patterns in the development of software applications and to develop pervasive pattern-level views that document a design as a composition of patterns. Patterns tend to be lost and blurred at the implementation and low-level design phases. The proposed POAD process and the associated UML models provide a solution for this problem. We discuss the support of the Unified Modeling Language to model pattern-oriented designs. We illustrate the use of UML modeling capabilities to develop three logical views, *Pattern-Level*, *Pattern-Level with Interfaces*, and *Detailed Pattern-Level* to facilitate the process of designing with patterns. The three views are based on the principle of pattern interfaces and support hierarchical traceable designs where high-level views of collaborating patterns are traced to lower level views of collaborating classes. One challenge to the POAD approach is how to analyze the user requirements for the purpose of selecting patterns. Moreover, we did not address the problem of how patterns can be combined with parts of design that are not expressed as patterns. Several applications may include application classes or frameworks as building blocks. Since we are using UML models, we expect that other modeling construct could be directly integrated with POAD models.

References

1. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, 1999.
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Object-Oriented Software. Addison-Wesley, 1995.
3. Vlissides, J.: Pattern Hatching, Design Patterns Applied. Addison-Wesley, 1998.
4. Castellani, X., and S. Y. Liao, "Development Process for the Creation and Reuse of Object-Oriented Generic Applications and Components", *Journal of Object Oriented Programming*, June 1998, Vol 11, No.3, pp24-31
5. D'Souze, D., and A. Wills. Objects, Components, and Frameworks with UML. Addison Wesley 1999.
6. D'Souza, D. "Interface Specification, Refinement, and Design with UML/Catalysis", *Journal of Object Oriented Programming, June 1998, pp12-18*
7. Alexander, C., S. Inshikawa, M. Silverstiein, M. Jacobson, I. Fiksdahl-king, and S. Angel, “A Pattern Language”, Oxford University Press, New York, 1977
8. Schmidt, D. : Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In Pattern Languages of Program Design, Coplien, J. and Schmidt, D. (eds.), 1995, Chapter 29, pp529-545.
9. Distefano, J., A. Stubberud, and I. Williams, “Feedback and Control Systems”, McGraw-Hill, 1990
10. Yacoub, S., Ammar, H.: Towards Pattern Oriented Frameworks. The Journal of Object Oriented Programming, JOOP, January 2000.

11. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture - A System of Patterns. Addison-Wesley, 1996.
12. Martin, R., Riehle, D., Buschmann, F. (eds.): Pattern Language of Program Design 3. Addison-Wesley, 1998.
13. The Unified Modeling Language homepage. http://www.omg.com/technology/uml
14. Pree, W.: Design Patterns for Object-Oriented Software Development. Addison-Wesley 1995.
15. Garlow, J., Holmes, C., Mowbary, T.: Applying Design Patterns in UML. Rose Architect, Vol 1, No. 2, Winter 1999
16. Fowler, M.: Analysis Patterns. Addison Wesley, 1997.
17. Yacoub, S. and H. Ammar. Pattern-Oriented Analysis and Design. Addison Wesley, to appear 2002.
18. Riehle, D. Composite Design Patterns. *Proceedings of Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA'97, pp218-228, Atlanta, October 1997.
19. Kent, S. Constraint Diagrams: Visualizing Invariants in Object-Oriented Models. *Proceedings of Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA'97, Atlanta Georgia USA, October 1997.
20. Lauder, A., and S. Kent. Precise Visual Specification of Design Patterns. *Proceedings of the 12th European Conference on Object Oriented Programming*, ECOOP'98, pp114-134, Brussels, Belgium, July 1998.
21. Keller, R., and R. Schauer. Design Components: Towards Software Composition at the Design Level. *Proceedings of 20th International Conference on Software Engineering*, ICSE'98, pp302-311, Kyoto, Japan, April 19-25, 1998.
22. Schauer, R., and R. Keller. Pattern Visualization for Software Comprehension. *Proceedings of the 6th International Workshop on Program Comprehension*, (IWPC'98), pages 4-12, Ischia, Italy, June 1998.
23. Larsen, G. Designing Component-Based Frameworks using Patterns in the UML. *Communications of the ACM*, 42(10):38:45, October 1999.
24. Rogers, G., "Framework-Based Software Development in C++", Prentice Hall 1997
25. Soukup, J.. Implementing Patterns. Chapter 20, pp395-415, in *Pattern Language of Program Design*, Addison-Wesley, 1995.
26. Bosch, J. Specifying Frameworks and Design Patterns as Architecture Fragments. *Proceedings of Technology of Object-Oriented Languages and Systems*, China, Sept. 22-25 1998.
27. Srinivasan, S., and J. Vergo. Object-Oriented Reuse: Experience in Developing a Framework for Speech Recognition Applications. *Proceedings of 20th International Conference on Software Engineering*, ICSE'98, pp322-330, Kyoto, Japan, April 19-25, 1998.
28 Wirfs-Brock, R., and B. Wilkerson. Object-Oriented Design: A Responsibility-Driven Approach. *Proceedings of Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA'89, pp71-75, October 1989.
29. Reenskaug, T.OORASS: Seamless Support for the Creation and Maintenance of Object Oriented Systems. *Journal of Object Oriented Programming*, 5(6):27-41, October 1992.
30. Reenskaug, T.. Working with Objects: The OOram Software Engineering Method. Manning Publishing Co., ISBN 1-884777-10-4, 1996.
31. Yacoub S., and H. Ammar, "UML Support for Constructional Design Patterns", in the special issue of *The Object* journal on Object Modeling, Hermes Science Publications, B. Henderson-Sellers nad F. Barbier (eds.) 2000.