Conversion Strategy of a System of Collaborating Design Patterns into UML Diagram for Design Pattern

Muhammad Mustapha¹, Nik Ghazali Nik Daud¹, Indika Udawatta², Manjula Atapattu²

¹ National Defense University of Malaysia, Sungai Besi Camp, 57000 Kuala Lumpur, Malaysia; {muhazam, nikghazali}@upnm.edu.my
² Maharishi University of Management, Fairfield, Iowa 52557, USA; {indikau, manjulaatapattu}@gmail.com

ABSTRACT

In a past publication, a proposal of a new set of UML diagram designed to picture a software system at architectural level by showing which patterns are being used has been presented. Progressing from the proposal, this publication presents an extensive example to precisely show the way to make the conversion. In this paper we present the proof of the ability of the proposed diagram set to professionally and neatly picture the overall high level architecture of a software system. There is also an additional rule added to the one presented in the past publication. This new rule is to cater for the cases where collaborating patterns are not overlapping; rather they are related just because some class in one pattern is using some other class in the other pattern.

KEYWORDS

Design Patterns, Unified Modeling Language (UML), Architecture Description Language (ADL), Object Modeling Language (OML), Object Oriented Programming (OOP)

1 INTRODUCTION

1.1 UML and Design Pattern Background

One can agree that at the highest level of system design, design pattern [1] has been adopted as a way to model and communicate software ideas and architecture [2] [3] [4] and, more importantly, to start the design idea not-from-scratch [5] [6].

One of the most common ways to represent design pattern is through the use of unified modeling language (UML). However, unknown to many, the de facto inventor of design pattern, Gamma et al. did not represent the design patterns in their book [1] in UML format, but in object modeling technique (OMT) instead.

UML has been around since 1997, and as of now it comprises of 14 diagram sets [7] [8] [9]. Despite the fact that it has been used widely in program development, and despite the fact that it
has so many diagrams – that may seem to be redundant – UML is still being regarded as incomplete [7]. Due to this shortcoming, there are numerous works have been done to represent design patterns in form of existing UML diagrams. For example, Dong et al. [10] [11] [12] [13], Le Guennec et al. [14], and many other significant works [15] [16] [17], have performed extensive studies to achieve such visualization with UML features. The infrastructures that have been used include annotated package diagrams, collaboration diagram, roles, tags, stereotypes, metamodel profiling, as well as Venn-diagram-like dotted or shaded area, object constraint language [18] and Object-Z [19].

1.2 Placing This Paper into the Background

Jakubík [2] proposed almost the same idea of representing design patterns with new kind of diagrams. Besides that, Yacoub and Ammar have used interface diagrams in [20] [21] to illustrate a way to represent design patterns. These three articles have almost presented the very idea that is to be delivered in this paper. However, their main purpose wasn't to develop a new type of diagram that has potential to be part of UML.

Besides the three papers, Zdun [22] [23] and Buschmann et al. [24] have also presented quite a similar idea but more on expressive languages rather than diagrams.

This gap that has been left behind by these six closest articles is the one that our past publication [25] had filled in. In that paper we have presented a proposal of a new diagram set for UML dedicated to picture design patterns instead of indirectly picturing design pattern using other existing diagrams. Brief conversion techniques are also given based on the pattern's structure and behavior.

Other than that, the paper was also focusing on simple, front-end sketch-type diagram that is to be used at inception phase of software development. To achieve it the paper laid out some connection rules to be used with the diagram.

Progressing from the proposal, in this paper we would like to present an example of a reduced real programming project whose architecture was laid out mostly on GoF's pattern. This is the main contribution of this paper. We will show how the traditional class diagram drawing of software architecture containing GoF's design patterns would be converted into the new dedicated pattern diagrams.

To simplify our writing, in this publication we will refer "dedicated UML diagram for design pattern" as "pattern diagram."

2 BACKGROUND INFORMATION OF THE EXAMPLE

Before starting with the discussion on the example that is to be presented in this paper, some background information from the previous paper will be given in
this section in order to maintain smooth transition of idea.
Figure 1. Example of Network of Class Diagrams Representing Architecture of a Software System
The network of class diagrams of the example is as shown in Figure 1. It has been drawn using Microsoft Visio hence some aspects of UML feature might not be shown. This example is a reduced version of a banking system application. It is impossible to present the entire original system due to:

i. the space constraint in this paper,
ii. copyright and confidentiality issues,
iii. some parts are platform-specific (Java), and,
iv. some patterns used are not GoF's patterns.

2.1 Overall Quality of Class Diagram Layout

The application architecture layout quality of the example in Figure 1 is fairly good after the removal of many structures, classes and patterns that are not relevant for the discussion in this paper.

Yet, it is still hard to detect the patterns used if the diagram only has class diagrams. To highlight the patterns, the diagram has to add extra boxes around them and label it with names. By adding such features, we might have highlighted the patterns but we are also interfering with the original key features in the diagram, especially the links, and making them hard to see. The same thing happens if we use the available UML infrastructures to highlight design patterns.

All design patterns in the diagram are able to be surrounded with rectangle boxes except observer pattern. The boundary of the observer pattern has to be irregularly shaped so that it won't overlap with classes in iterator pattern.

Nonetheless, this is in fact a good example where we can find all types of possible overlapping that can happen:

i. No overlap: Pattern iterator does not overlap with any other patterns
ii. Partial overlap: Pattern observer overlaps partially with template method and strategy patterns
iii. Full overlap (nested): Another strategy pattern is nested inside a decorator pattern

2.2 Design Patterns in Use

Figure 1 shows that there are 5 design patterns are being used in the example: iterator, observer, template method, decorator and strategy. We extract some part of our past publication [25] to show the partial conversion of the class diagram representing design pattern into the pattern diagram. The rationales of their use in the example are also given:

**Template Method Pattern:** The customer related classes (Customer, Company and Personal) are implemented as template method pattern. Unlike the Account class, whose children (Savings and Checking) are different in term of method strategies, the Customer's children are not strategically different. Rather, they are different in term of property content. Hence template method pattern is more suitable for these types of classes. Conversion of template method pattern:
Strategy Pattern: The Account and Entry classes are implemented as strategy pattern. The rationale for this so that new accounts can easily be added to the system and new kind of entries can easily be written into the account information. This can be done just by extending (inheriting) from the Account or Entry abstract classes. This addition would be done at compile-time, hence strategy pattern is the best (state pattern is better if the addition is at run-time.) The Entry abstract class itself is a part of decorator pattern that will be explained later. Conversion of strategy pattern:

Decorator Pattern: The way that the entry lines in the account are printed is done by partial decorator pattern. It is partial since no concrete class is extended from the top-most abstract class of LineEntry. However the structure of decorator pattern is kept for future addition of such concrete classes. The entries of withdraw, deposit, interest, etc are created as strategy pattern under Entry abstract class. This means the strategy pattern is a sub-pattern in the decorator pattern that forms the different decoration components. Decorator pattern conversion:
Observer Pattern: There are circumstances where the customers need to be informed about the activities in their accounts, like when a large transfer takes place. This scenario can be implemented using observer pattern. For the example in concern, the subject part of the pattern is the account, while the observer is the customer. Observer pattern conversion:

Iterator Pattern: Iterator pattern is for looping through all customers and accounts. This is normally done through data structures provided by the respective framework available in the platform used. Iterator pattern conversion:
Besides the above patterns, there are two more classes, BankCompany and BankingSystem, which are the main users (clients) of the system. These 2 classes are not part of any pattern, and together they will be considered as client class.

3 ADDITIONAL RULE

In the past publication [25] the only reason that a link is made between two pattern diagrams is because they overlap. This means the patterns share a particular class, so the link is put in order to highlight the repetition of the class in the two patterns.

It is learnt, however, that in many cases, there might not be overlap between two patterns, but they are still related to each other by usage; i.e. some class in one pattern might be using some class in the other pattern. For this reason, the original proposal of the pattern diagram will now be added with another infrastructure to highlight such connection. The infrastructure is an open headed arrow:

The example in Figure 7 shows that Class B in Pattern 1 uses Class E in Pattern 2 in some way. The usage link should follow the same connection rules as given before in [25].

Note: (1) It is important to bear in mind that showing usage relationship is not important at inception phase. The same objective can actually be achieved by putting remark labels on those class names just like we normally add to class diagrams. As a matter of fact, during inception phase, "islands" of patterns (a group of patterns that are linked to each other but not to the others in the system) are acceptable and more preferable since it allows the diagram to be broken into few parts. (2) The link due to overlaps (link with no open head) overrules the link due to usage. It is obvious that if the patterns overlap then there must be some kind of usage exists.

4 THE CONVERSION

Now we will convert the given class diagram network in Figure 1 part by part into the proposed pattern diagram.
4.1 The Customer Classes Hierarchy

The hierarchy of customer classes, involving class Customer, Company and Personal, as mentioned before, constitutes template method pattern. Since the 2 concrete classes share the same abstract class they would be listed in the same diagram:

4.2 The Account Classes Hierarchy

The hierarchy of account classes, involving class Account, Savings and Checking, forms strategy pattern:

4.3 The Account Entry Classes Hierarchy

Excluding the abstract class LineEntry, the hierarchy of account entry classes, involving class Entry, Withdraw, Deposit and AddInterest, also forms strategy pattern:
4.4 The Line Entries Aggregate

The abstract class LineEntry, together with the entry classes, form decorator pattern. The differences among the decoration lines in printing will be the differences in terms of the strategy they calculate the respective values. Even though this pattern doesn't have the concrete component as in GoF's definition of decorator pattern (Section 2.2), they still form a proper decorator pattern as the concrete components are not compulsory. The complete conversion of this pattern will be shown later when we link it with the nested strategy pattern. The partial conversion is:

```
+print()
| Amount
| Date
| Description
```

4.5 Observer Hierarchy

The three classes that constitute observer pattern are the three abstract classes of Observer, Customer and Account. Notice that the Account class acts as subject in the pattern but there is no subject abstract class that matches the one in GoF's definition. The reason why the architecture was initially designed that way is unknown.

However, everything that the Account class needs to act as subject is given in its own declaration and definition. Without the dotted border line, this contract (agreed requirement to act as subject) is not obvious in the original class diagram, but in the pattern diagram, it is more obvious. The word "subject" is put in the box as stereotype where the abstract class Subject is supposed to be. This is to show that even though there is no Subject abstract class, the abstract class Account still needs to implement everything that the missing class has to do. This scenario shows that while allowing flexibility in pattern implementation, the dedicated diagram still can clearly highlight the contract that the classes have to adhere to in order to be recognized as that particular pattern.

4.6 Iterator Hierarchy
Notice that in the original class diagram, there are two iterators: one with `TreeMap` as aggregate and the other one is with `AList`. It is not clear in the class diagram, but `TreeMap` is used by the top-most class (`BankCompany`) to keep a list of customers and accounts in `AList`.

In the solution pattern diagram, the relationship between the two iterators will not be shown; not even using the usage link (open headed link). The reason is because the two patterns actually belong to a single pattern. Hence they will be drawn in one diagram.

Another point worth highlighting in this part is the abstract `Iterable` class. The reason for using this class is because the actual platform used to develop the system was Java that requires inheritance from the class. In the UML diagram we will add this class together with `Collection` class to play the role as aggregate and a check mark [13] will be added to show that the class is provided by a specific framework (J2EE).

![Diagram](image)

**Figure 13.** Iterator Conversion

### 4.7 Linking the Diagrams

The complete network of pattern diagram is obtained by linking the pieces in the previous sub-sections. Connect the diagrams that overlap by joining the shared classes. Optionally we can also connect the name of the classes that use other classes across pattern boundary using open headed arrow (Section 3).

In this example we will add another leeway by allowing the top class client to be added in the final diagram as `class` diagram not belonging to any pattern. These top most classes are the classes of `BankCompany` and `BankingSystem`.

Normally in UML diagrams, if it is of a particular diagram, other types of diagrams are not allowed. For example, in a state diagram no class diagram is allowed. However that is not always the rule.

The final pattern diagram solutions for the example are in Figure 14 (with usage links) and Figure 15 (without usage links). In Figure 15 it can be seen that Patterns Template Method, Observer and Strategy form an island, another Strategy Pattern and Decorator Pattern form another island, and the Iterator Pattern
forms another island by itself. The clients class is also not shown in this figure as they are not important inception phase.

Figure 14. Conversion to Pattern Diagram with usage links
5 CONCLUSION

It is obvious in this lengthy discussion of the example that the pattern diagram offers a more architect friendly diagram compared to class diagram. The solution in Figure 14 and 15 can easily be fit into half a page in portrait orientation, while the original example in Figure 1 is not so easy to put in one page and have to be drawn in landscape orientation.

Skeptics of this pattern diagram may want to raise questions about the information that the diagram hides that is normally visible in class diagrams. As we have stressed this a few times in the past publication, the proposed pattern diagram is for inception phase. In this phase we just want to see the overview of the overall system, and as much as possible we try to start the architecture directly using design patterns. The details inside each pattern would be filled in by the subgroups of programmers assigned to code the respective patterns. Hence, the issues about details are not relevant for the pattern diagrams.

6 REFERENCES