How Does the Implementation of the Observer Pattern in Freecol Affect Reusability, Extendibility and Flexibility of the Application as measured by CBO?

Tova Linder
TDDB84 Design Patterns
Linköping University
Linköping, Sweden
Email: tovli744@student.liu.se

Abstract—Design patterns are widely used to solve common design problems. This paper analyzes the use of the observer design pattern, more specifically as implemented in the Java application FreeCol. The paper will focus on the software quality characteristics reusability, extendibility and flexibility. It defines what software quality is and how reusability, extendibility and flexibility can be measured. The paper further analyzes different methods and propose CBO as a proper metric to evaluate these software quality characteristics. The paper also discusses how the observer design pattern is implemented in FreeCol and if (and how) this implementation benefits FreeCol. Lastly some setbacks of using the observer pattern are found and alternatives for handling communication between objects in these cases are suggested.

I. INTRODUCTION

The world today relies on software and it is highly important that software works properly. Software quality has to be ensured, but what is software quality? According to Chapell [1] it can be split into three main areas, structural (code structure), process (development process) and functional (functionality of final product). What type of quality that matters the most depends on who you are. While a user probably is most concerned about the functional quality, developers who are responsible for maintaining the application probably are more interested in the structural and the process quality. These three qualities are closely linked together, improving one might improve or worsen another [1].

When considering the field of design patterns the study is limited to structural quality since design patterns handle code structure. Gamma et. al. [2] presents design patterns as a common structure to solve a repeatedly occurring problem even if the implementation itself differs. When object oriented (OO) design was first introduced it opened up a new way of designing our software systems. The usefulness of design patterns has been questioned repeatedly since then. But as Chandra and Linda declare in [3] “simply by using OO language or design does not assure a robust and reusable system. It mainly depends on the pattern or inter-dependencies between the subsystems and the communication between them”, there is certainly a need for structural practices to exist. Although Zhang et. al. show in [4] that there is no evidence that design patterns will increase program structure they do find some support that design patterns provide a good framework for maintenance.

The target of the analysis is FreeCol, a turn based coloniztion game. It is an open source application written in Java [5]. This paper will analyze how the implementation of the observer design pattern in Freecol affects the reusability, extendibility and flexibility of the application. This is done by studying a sample implementation from the application and by referring to literature. Proper methods for evaluating reusability, extendibility and flexibility of an application are discussed and proposed.

The rest of this paper is organized as follows, section II covers background theory, section III presents the implementation of the observer design pattern in Freecol, section IV analyzes the implementation in FreeCol and the use of the observer design pattern in general and finally in section V the conclusions are presented.

II. BACKGROUND

This section will cover background theory and definitions of the observer design pattern and the evaluation techniques that are to be used in this paper.

A. The Observer Design Pattern

The observer design pattern is categorized as a behavioural design pattern in Design Patterns: Elements of Reusable Object-Oriented Software [2]. The pattern is useful when an object have an one-to-many dependency to other objects. The idea is that when the subject (the object that others are dependent on) changes, updates are sent to all the dependents, called observers. In object oriented programming we strive to break down large applications into many cooperating classes [2]. An example is the separation of the graphic interface and the data being modified in the underlying system. When that approach is taken and we still want to maintain the relationships between
objects it can result in a very high coupling between the classes in the application. Using the observer pattern reduces the coupling [2]. The reduced coupling allows for Open Closed principle.

A method called QMOOD presented by Bansiya et al. in the paper A Hierarchical Model for Object-Oriented Design Quality Assessment [8] is based on the characteristics in the ISO/IEC 9126 model. After some exclusion, replacement and inclusion of characteristics they end up with some characteristics they consider relevant for measuring software quality: functionality, effectiveness, understandability, extensibility, reusability and flexibility. Reusability is defined as the presence of an OO design that can be reapplied to a new problem without significant changes. They further present extendibility as the possibility to add new requirements into an existing design and flexibility as the possibility to make changes in the current design. In order to measure these characteristics they find components upon which definitions are built up. Reusability is according to [8] defined as a linear combination of coupling, cohesion, messaging and design size. Extensibility is a linear combination of abstraction, coupling, inheritance and polymorphism and flexibility is a linear combination of encapsulation, coupling, composition and polymorphism.

Software quality can be measured in many ways. One way could for example be to simply let test persons add or change functionality to the existing application and by evaluating their experience find how extendable and flexible the application is. Or let them apply the same code to a new problem to figure out how reusable the code is.

More structured metrics could also be used. An important factor when looking at reusability, extendibility and flexibility of an application is how closely connected different objects in the application are, also called coupling [8]. This can be measured by a metric called CBO (Coupling Between Classes) which was developed by Chidamber et al. [9] in 1994 for measuring quality of object oriented software designs. Many previous metrics for measuring software quality had not been suitable for object oriented design. The CBO is a measure of how many classes a specific class is directly related to [9] [10]. The same metric is also presented in [8] as DCC (Direct Class Coupling). A low coupling, which means a low CBO value, is desirable to achieve an effective design that is more easy to understand, reuse and extend [10].

There are also suitable metrics for the rest of the elements representing reusability, extendability and flexibility. For example abstraction can be measured with ANA (Average Number of Ancestors), inheritance with MFA (Measure of Functional Abstraction), polymorphism with NOP (Number of Polymorphic methods), encapsulation with DAM (Data Access Metric) and composition with MOA (Measure of Aggregation) [8]. CBO is chosen for further analysis since it measures coupling and all three characteristics (reusability, extendibility and flexibility) are dependent on coupling. Also Bakota et al. [11] studied how effective different types of metrics are for general systems and found that CBO behaves similarly on all systems and therefore is suitable for measuring coupling in a general model.

1) Coupling: Coupling is defined in [9] as class X is coupled to a class Y if at least one of them acts upon each other, in order words if one of the classes can affect the other class’ history. Let $X = < p(x) >$ and $Y = < p(y) >$ be the two objects and $p(x) = \{ M_x \} \cup \{ I_x \}$ $p(y) = \{ M_y \} \cup \{ I_y \}$ where $M$ represents methods and $I$ instance variables. This definition of coupling gives us that an action by $\{ M_x \}$ on $\{ M_y \}$ or $\{ I_y \}$ and the other way around ($\{ M_y \}$ on $\{ M_x \}$ or $\{ I_x \}$) result in object A and B being coupled [9].

Coupling serves as a definition for all types of interactions between objects but in [12] but Offutt et al split this definition to a few different types of coupling:

- Parameter coupling - method calls between classes.
- External/file coupling - classes that access the same external medium, for example files.
- Inheritance coupling - a subclass that uses a non-redefined variable from its superclass.
- Global coupling - variable defined in one class but used by other classes.

2) CBO: CBO considers all types of coupling described in previous section except for external/file coupling and simply counts the number of other classes one class is coupled to by any of these means [9]. These are all other types that are used in attributes, parameters, return types, throws declarations etc.. Primitive types and systems types (java.lang.*) are not counted. But how does one know what an acceptable value for CBO is? There is no consensus for software metrics thresholds [13] which means there is no general threshold that can be used.

In a study [3] a method for evaluation class design validity called Class Break Point is presented. The method uses the
Chidamber and Kemerer (CK) metrics, which are the metrics presented in [9], and one of these metrics is CBO. They suggest a threshold for CBO that is 0-8. If the value is not in that range they suggest design refinements to improve reusability [3]. One study, [13], found that a threshold value for low/high risk of bugs is CBO=9. The same study also claims that many other studies have found that CBO is a significant software quality factor [13]. Based on these studies this paper considers a value in the range 0-9 as an acceptable threshold value.

III. IMPLEMENTATION IN FREECOL

It is very common to use the observer pattern when creating applications containing a graphical user interface that has to react to changes of data in underlying classes. A more specific implementation of the observer pattern is as part of a Model-View-Controller (MVC) structure [2]. In this structure the model serves as the subject containing data that can change over time. Every time the model changes it reacts by notifying the views to update themselves [2]. The controller is not a part of the observer pattern since it handles the user input, but it is important since it is commonly a user that triggers actions. There are many examples of listeners implemented from the java standard library in FreeCol, such as mouse and keyboard listeners. For this analysis however a FreeCol specific implementation of the observer pattern is chosen.

A. Transaction Listener

In FreeCol the observer design pattern is used to control the logging of sales and purchases on the market. First the design is presented in an UML diagram and below there are relevant sections from the code.

```java
public final class TransactionLog extends JTextPane implements TransactionListener{

    public TransactionLog() {
        setEditable(false);
    }

    public void initialize() {
        getMyPlayer().getMarket().
            addTransactionListener(this);
        setText("");
    }

    public void cleanup() {
        getMyPlayer().getMarket().
            removeTransactionListener(this);
    }

    public void logPurchase(GoodsType goodsType
        , int amount, int price) {
        ...
    }

    public void logSale(GoodsType goodsType,
        int amount, int price, int tax) {
        ...
    }

    private void add(String text) {
        ...
    }
}
```

```java
public final class TransactionListener {

    public void logPurchase(GoodsType goodsType,
        int amount, int price);

    public void logSale(GoodsType goodsType, int
        amount, int price, int tax);
}
```

client.gui.panel.(EuropePanel//TransactionLog.java

```java
public final class TransactionLog extends
    JTextPane implements TransactionListener{

    public TransactionLog() {
        setEditable(false);
    }

    public void initialize() {
        getMyPlayer().getMarket().
            addTransactionListener(this);
        setText("");
    }

    public void cleanup() {
        getMyPlayer().getMarket().
            removeTransactionListener(this);
    }

    public void logPurchase(GoodsType goodsType
        , int amount, int price) {
        ...
    }

    public void logSale(GoodsType goodsType,
        int amount, int price, int tax) {
        ...
    }

    private void add(String text) {
        ...
    }
}
```

common.model.Market.java

```java
public final class Market extends
    FreeColGameObject implements Ownable {

    private ArrayList<TransactionListener> transactionListeners =
        new ArrayList<TransactionListener>();

    public void addTransactionListener( TransactionListener listener) {
        transactionListeners.add(listener);
    }

    public void removeTransactionListener( TransactionListener listener) {
        transactionListeners.remove(listener);
    }

    public TransactionListener[]
        getTransactionListener() {
        return transactionListeners.toArray(new TransactionListener[0]);
    }

    ...
```

common.model.TransactionListener.java

```java
public interface TransactionListener {

    public void logPurchase(GoodsType goodsType,
        int amount, int price);

    public void logSale(GoodsType goodsType, int
        amount, int price, int tax);
}
```

```java
public interface TransactionListener {

    public void logPurchase(GoodsType goodsType,
        int amount, int price);

    public void logSale(GoodsType goodsType, int
        amount, int price, int tax);
}
```
IV. ANALYSIS AND DISCUSSION

A. Analysis of implementations

In the canonical representation of the observer pattern described in figure 1 it can be observed that the subject has a method for notifying its observers. In the implementation of the pattern in FreeCol this has been solved with a slightly different approach. Instead of letting the subject itself notify all the observers the subject has a method which returns all listeners connected to it. By using this method any other object can access the list of listeners and notify them by running one or several of their methods.

In this paper coupling is the most important element of analysis as presented in section II-B1. It is also specifically interesting to focus on coupling since the intent when introducing the observer design pattern is to minimize the dependencies between communicating classes. Using an observer pattern creates a minimal coupling between the subject and the observer [2].

In FreeCol the observer pattern plays an important role for the graphical user interface. To achieve objects holding data that are decoupled from the actual view there are many uses of Listeners (such as mouse and keyboard listeners) that are directly implemented from the java standard library. These implementations are important since they decouple the GUI from the data in the program.

The use of observer pattern does benefit both the extensibility and the flexibility of the application. Examples of extending the application can be addition of a new part in the GUI that needs to be notify about changes on the market. This class can easily be added and just by implementing the TransactionListener and be registered at the subject it can receive the updates just like any other class that implements the interface. Flexibility can be achieved through the possibility to register and deregister listeners to an observer at any time in the application.

It has been shown [10] that low coupling provides a modular design that is easier to reuse and apply to other similar problems. By introducing the observer pattern the coupling is reduced which means that the observer pattern does benefit the reusability of the code. For example could the market class easily be changed by replacing the array of TransactionListeners with some other listener and with very few modifications that class would still function properly.

B. Analysis of CBO

To calculate CBO for the different classes involved in the design pattern av tool called ckjm\(^1\) is used. The result for the different classes involved in the design is presented below. The reason for including EuropePanel.java in the table is that TransactionLog.java is an inner class in that class. This way the TransactionLog object can access all members of its outer class EuropePanel.java. An inner class is a way to logically group classes that are only used in one place, it increases the encapsulation and leads to more readable and maintainable code [14]. This differs a little bit from the canonical representation of the observer pattern but if the TransactionLog class was used by more than one class it could easily be extracted to a separate class.

<table>
<thead>
<tr>
<th>Class</th>
<th>CBO</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransactionListener.ja</td>
<td>1</td>
</tr>
<tr>
<td>Market.java</td>
<td>14</td>
</tr>
<tr>
<td>TransactionLog.java</td>
<td>9</td>
</tr>
<tr>
<td>EuropePanel.java</td>
<td>30</td>
</tr>
</tbody>
</table>

The CBO value for TransactionLog and TransactionListener is just around the threshold value and far below it respectively. The threshold values are determined in section II-B2. However, the validity of the threshold values presented in [3] and [13] should be considered with some care since these are only two separate studies with very specific research topics. The fact that two separate papers have found reasonably similar threshold values does although support that these values might be relevant also for this paper. The Market class has a CBO value that is relatively high but this value is mainly not created by the observer and the subject in the observer pattern but from surrounding code. One way of realizing the advantage of having the observer pattern introduced is to think of an implementation where every class that require updates on changes on the market had to be coupled to the Market.java class. Such implementation would primarily increase the CBO value when first creating the application and secondarily it would be very harmful to extend with new classes that requires updates since the CBO value would increase by one for every introduction of a new class.

As a comparison of the CBO values seen in FreeCol the analysis presented in [10] can be used. They use JFreeChart\(^2\), which is the most widely used chart library for Java, as a target to conduct measurements of CBO and some other metrics. JFreeChart is of similar size as FreeCol:
- JFreeChart (v.1.0.14): 667 files, 224395 lines of code.
- FreeCol: 669 files, 192709 lines of code.

The authors found that many of the metrics indicated a declining software quality over time as the authors analyzed 21 different versions of the library. Most interesting was that the average CBO for the first analyzed version was 9 and for the latest analyzed version (1.0.14) was 15.28. These values lie reasonably close to the calculated values for the FreeCol classes that were analyzed.

C. Should observer design pattern be used?

The effectiveness of using design patterns has been questioned by many. Prechelt et. al. presents in [15] a few trade offs experienced when using design pattern. Advantages pointed out are that design pattern gives a common terminology

\(^1\)ckjm, http://www.spinellis.gr/sw/ckjm

\(^2\)JFreeChart, www.jfree.org/jfreechart/
and that design patterns follow best practice. The study was conducted by comparing a design pattern solution to a simpler alternative solution to a design problem. The negative sides they find are that code can get more complicated which can effect understanding of the code which makes it less likely that the code can be reused (reusability) and makes changes to current design more complicated to perform (flexibility and extendibility) [15]. Similar advantages and disadvantages are pointed out by Zhang et. al. [4] in a systematic literature study of many other papers about the use of all 23 patterns presented in [2]. They find that design patterns can help improve communication between developers and create a common vocabulary. They also recognize that if design patterns are implemented and not well documented their presence can make it harder to understand the code.

The experiments performed in [15] are among other patterns studying specifically the observer design pattern. They find that the observer design pattern is often introduced to solve a design problem which does not necessarily require the full functionality of the pattern. This type of implementation was found to be harmful when later wanting to extend the application. Others [16] argue that the applications that are used today have developed from static applications to highly interactive once. While this has happened the observer based event handling is still used in many production environments despite its error prone implementations. They instead propose a framework for reactive programming [16].

The general disadvantage of using the observer to achieve loose coupling is that the subjects and the observers often are connected to each other when they are created, sometimes it can be hard to understand the implementation of the observer pattern. The use of frameworks is one common solution to handle large applications with many observers [2].

V. CONCLUSIONS

Software quality is a very ambiguous concept and there are many different models that propose a slightly different approach to what software quality is. Many of the models resemble each other by the use of similar software characteristics on which software evaluation are based upon.

Reusability, extendibility and flexibility can be measured either by user evaluations or by more sophisticated metrics. For all these three software characteristics coupling is an important factor, to measure coupling the metric CBO can be used. Researchers point out that there is no consensus on what is considered an acceptable CBO value but there are studies that suggest a CBO value between 0-9 as an acceptable value.

In FreeCol the observer pattern is, as in many other applications, widely used to decouple GUI objects from its underlying data classes. The pattern benefits FreeCol by reducing the coupling between classes and as for the implementation presented in this paper it has been found that most of the involved classes have a CBO value within acceptable limits. The loose coupling creates less error prone code which is easier to reuse, change and extend.

The paper also shows that many argue against the use of the observer pattern since it can make code complicated to read, especially if the number of observers increase. This is something that contradicts the use of the observer pattern and one recommendation is to head towards using frameworks that can handle the reactive programming instead.

ACKNOWLEDGEMENTS

The author would like to thank Arvid Söderström, Johanna Thorheim, Henning Hall, Niklas Ljungberg och Adam Sestorp for great cooperation throughout the course. A special thanks to Arvid and Johanna who have reviewed this paper and added valuable comments to the work.

REFERENCES

[2] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1995.
Improvements After Review:
How Does the Implementation of the Observer Pattern in Freecol Affect Reusability, Extendibility and Flexibility of the Application as measured by CBO?

I. INDEPENDENCE
Both my reviewers put a 4 here and no specific comments on what would need to be changed. Since a 5 requires an advanced topic and I’m satisfied with the references I have used I have only added some more references that were required to satisfy other comments from the reviewers.

II. TOPIC
The reviewers commented that it was very clear what corresponds to X, Y, Z and W. They also said that the topic and definitions were well described. For topic I have not changed much.

III. LANGUAGE AND FORM
I got some comments on language. The reviewers pointed out that I should not use "we" when talking about the study but instead use "the paper" or similar expressions to get a more formal paper. This has been modified in the entire paper. They also pointed out a few spelling errors and grammatical mistakes which I have corrected. In some places I had used American spelling and some British, this is also changed to be consistent throughout the report.

IV. SOURCE MANAGEMENT
I have discussed citations with my reviewers and one of them had comments on where the citations in the text were placed. I had to find some answers on how IEEE style citations in the text should be done. These answers made me change the location of the citation in many places in the text.

I also got comments on the use of images and that is was unclear which images I have created myself and which I have downloaded. To clarify this I added a citation to the images that are not mine.

The reviewers also found some paragraphs that lacked a reference, either because I just had missed to put it there (but I have the intended reference in the reference list) or because I just had no reference to justify my claim. This has been improved by adding references where I considered it needed.

Finally I got the comment that I had to some extent evaluated some sources. To improve this I have now tried to evaluate some more references and make it more clear in the text.

V. ANALYSIS
Since I was finished with my report for the seminar deadline I didn’t get that many comments on it.

I got a comment on my analysis stating that I don’t compare two different FreeCol code snippets, one with the design pattern implemented and one without. This was something we had to ask the examiner about. We concluded that the criteria is met even with different comparisons (than comparing one FreeCol implementation with a certain design pattern and one without the pattern) such as mine. Therefore this comment has not led to any major changes in the text.

Other than that I got positive comments on my analysis and only small changes has been done since last deadline.