The relationship between software quality and the use of the Command pattern

Abstract
This paper covers the relationship between software qualities and the command pattern. Two different open source projects, FreeCol [5] and Open-IG [6], will be analysed with respect to two different implementations of the command pattern found in their code bases. The pattern will be related to relevant software qualities to give an indication of what the pattern contributes with.

Design patterns are templates for supposedly good solutions to frequently appearing problems. The usefulness of design patterns is often taken for granted, and not much empirical research support the fact that these patterns are always helpful [2]. A way to analyse the correctness of this common assumption is to look at relevant software qualities, such as maintainability or extendibility, of a code base using these patterns.

The actual questions that this paper aims to answer are:
1. What can be good software quality factors for measuring the usefulness of a design pattern and how can these be measured?
2. Does the command pattern appear to be beneficial judging by how these relevant software qualities are affected?
3. Is it probable that the command pattern is almost always beneficial or harmful, or is it dependent on the code base in which they are used?

Introduction
Since the Gang of Four introduced their book “Design Patterns: Elements of Reusable Object-Oriented Software” [1] design patterns have become recognized as useful tools for solving commonly occurring problems in programming projects of any size. Their intention is to provide a template for solutions that are supposed to be generally applicable. While the actual implementations may differ the idea is constant over all domains. An example is the iterator pattern that allows for an effective solution when iterating over some structure by having the iterable structure itself define the way it picks the next element when being iterated over. It is a generally accepted fact that design patterns are beneficial to developers and users alike as the patterns typically provide a more neat solution than the alternative. Not much actual testing has been done on the topic, however, and thus there is no proof that the aforementioned claims are true. This paper intends to give some insight into this topic by attempting to compare some design pattern in real code with relevant software quality factors. Quality factor is here defined as being a particular subset of the complete quality of some piece of code.

Software quality is a topic concerned with measuring the quality of code. What makes code good is of course not obvious as one could look at a piece of code from many different viewpoints depending on what quality factor is of highest importance at the moment. A common way to look at software quality is therefore to divide the quality into many sub-areas that are treated and viewed individually [4]. A few of these common partitions of software quality are for example testability, maintainability and changeability. These factors concern very different aspects of the code so it makes sense to treat them separately.

For this particular paper not all quality factors will be of interest. This is not only because treating all aspects of the code would be too big of a task but also because design patterns likely are only intended to improve a subset of all the quality factors. Therefore it seems reasonable to restrict
oneself to only a few specific aspects of the broad term software quality when trying to judge whether design patterns are beneficial or not. How to accurately measure these quality factors is not obvious however.

**Regarding the use and measureability of design patterns**

As previously stated it is highly unlikely that a design pattern improves all aspects of the software’s quality and thus it would be unfair to judge a design pattern only because of a few negative side effects, given that there are some positive side effects that may outweigh the bad ones.

Some quality factors are only loosely affected by the pattern and are therefore uninteresting to check when trying to judge the pattern, portability is an example of such factor as how portable the code is probably has little to do with most implemented design patterns. Then there are likely a good amount of factors that are more directly affected by the implemented pattern. While it might be interesting to use many of these it is infeasible due to the size of such a task. Instead it appears as a good idea to narrow the scope and focus more on a few especially relevant quality factors, if it is possible to find such.

According to one paper [2] design patterns are primarily supposed to alleviate the difficulty of maintainability of the source code that implements the pattern. Rather than a completely arbitrary choice of quality factors this paper therefore restricts itself to primarily investigating design patterns effect on maintainability.

A problem with this approach is that maintainability can be hard to measure as it is a rather subjective quality. There exists some models that describe what makes up the different quality factors however. One early such model is McCall's quality model and another is the ISO 9162 which is a standard for software quality measurement. Both of these have in common that they consist of some top-level quality factors or characteristics that each are composed of several more fine traits [3]. The problem still remains though that these traits are rather subjective. For lack of a better way to measure the same traits for maintainability as those used in the ISO 9162 will be used. Maintainability will therefore here be measured using a linear combination of the code's analyzability, changeability, stability and testability.

It would be useful to complement this analysis with some actual measurements. It is not obvious which of all possible metrics, where a metric is defined as a combination of measurements, are related to the code's maintainability. One way to measure the testability, which is an integral part of the maintainability, is proposed in a paper on the topic of measuring and improving design patterns' testability [7]. The authors suggest that you measure the amount of class interactions and self usage interactions using an UML diagram of the implemented design pattern. A possible class interaction is defined as a class A having access to a class B through two or more different paths through the UML diagram, and a possible self usage interaction occurs when there exists a cycle from a class A to itself. The severity of the impact on testability is said to increase as the amount of class interactions and self usage interactions goes up. These two metrics will be used as a complement to the more subjective analysis of the different quality factors.

**Found implementations**

In FreeCol the command pattern was used to gather a lot of the actions pertaining to the GUI for easier access and modification later on. The Code below shows how a class adds all actions to a list that is later used to access and attach actions to for example JButtons. The second method shows such a use of the ActionManager.

```java
public class ActionManager{
    public void initializeActions(InGameController inGameController,
                                   ConnectController connectController) {
        add(new AboutAction(FreeColClient, gui));
    }
```
add(new AssignTradeRouteAction(FreeColClient, inGameController, gui));
add(new BuildColonyAction(FreeColClient, inGameController, gui));
add(new CenterAction(FreeColClient, gui));
...

public class EndTurnPanel extends JPanel {
    public EndTurnPanel(GUI gui) {
        ...
        add(new JButton(getFreeColClient().getActionManager().getFreeColAction(EndTurnAction.id)));
    }
}

public class EndTurnAction extends MapboardAction {
    public static final String id = "endTurnAction";
    private final InGameController inGameController;

    EndTurnAction(FreeColClient FreeColClient, InGameController inGameController, GUI gui) {
        super(FreeColClient, gui, id);
        this.inGameController = inGameController;
        setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_ENTER, 0));
    }

    public void actionPerformed(ActionEvent e) {
        inGameController.endTurn();
    }
}

In Open-IG a somewhat different approach was used. An abstract interface was implemented and used in many different and unrelated classes. In one case one class was both client and invoker while the receiver was an object local to the client. The use of the pattern in this case was thus essentially that of a queue. An example of this behaviour is shown in the code below.

In FreeCol all the actions were gathered in a single class for ease of access later on. This serves to

Analysis of maintainability
The patterns will first be analysed with respect to their effect on the codes’ respective maintainabilities. As mentioned earlier the maintainability can be measured as a linear combination of the different quality factors analyzability, changeability, stability and testability. The importance of each individual factor is not obvious and might well vary between software projects depending on what their goals are and what requirements they have.

In FreeCol all the actions were gathered in a single class for ease of access later on. This serves to
give a simple overview of what commands exist. It does not, however, say much about when or how the commands will be called upon. This is because the ActionManager is part of the GUI's client class which is present in a large amount of code. To get further insight into the uses of the commands one would potentially have to browse very large amounts of code which ought to impede the analyzability. That said, having all commands in separate classes and in their own package makes it easy to find out what they do, if not in what context. The found instances of the command pattern in Open-IG were all local to some specific class. That is command implementations were defined inside the client class as opposed to inside their own classes. While maybe not being as modular as the alternative this makes all code relevant to the client class accessible immediately and also decreases dependencies between classes which is a positive for someone wanting to analyze the code.

The changeability is defined as the effort required to change the code. This can in turn be roughly divided into the effort required to extend the code as well as replace the whole pattern with another solution. Extending the solution in FreeCol is as simple as creating a new method or reusing an existing one as it can be included anywhere at any time given that it follows the specification. More work would be required in Open-IG as all new implementations would have to be written from scratch. This could however have been an active choice from the developers as most different commands are likely only used once anyway, leading to no more code needing to be written. Replacing the whole pattern would require considerably more work, especially in FreeCol where the coupling between classes is tighter. But it is likely that the focus on the changeability factor lies more so on the extendibility part of it as a sufficient solution should typically not need replacing. It does however put some emphasis on the need to think things through before implementing the pattern as the cost of replacing it could be high.

The stability factor can be defined in several ways. The definition chosen here is the code's proneness to error after change, as that appears to be closely related to it's maintainability. The command pattern can not be considered to be especially vulnerable on this point as each new concrete command is not at all coupled with the others. With the framework already in place new errors can only be introduced by the new implementations. This is definitely the case in the aforementioned class in Open-IG since an addition or change to the existing cases only requires changing a few lines. The methods to be invoked are after all defined at the same places as where they are called upon. In FreeCol a change to an existing concrete command might have unintended effects if the developer is unaware of the context of each place where the command is called, which might realistically be the case in a project as large as this.

The testability, that is how difficult it is to test the correctness of the implementation, depends heavily on the complexity of the receiver. In the class in Open-IG the receiver is a somewhat complex implementation of an interface. While it might not be very easy to test this, though not necessarily hard either, this is not an effect of the command pattern. If anything the pattern is helpful in this aspect as it decouples the commands so that they can be tested separately. Here it might however be a drawback that the implementations are not written in separate class files as that would make testing more straightforward. In FreeCol an issue is that many actions have complex in parameters that make testing them a non-trivial issue. Still, in both cases one also has to consider the advantage of being able to test parts of the code in isolation even if it means having to initialize some unrelated objects.

**Measurements of testability**

In figure 1 some of the most relevant classes related to the implementation of the command pattern in FreeCol. One might note that there appears to be some rather complex relations among the classes judging by this class diagram. This in itself is not considered to be a fault when using the two concrete definitions, class interactions and self use interactions, mentioned earlier. An actual
error, however, appears to be a cycle between the ActionManager class and the FreeColClient class. The ActionManager is obviously central to this implementation of the command pattern since it is the only access point to it. This relation therefore classifies as a possible self use interaction caused by the command pattern.

![Figure 1. Class diagram showing classes in FreeCol related to the implemented command pattern.](image1)

The corresponding graph for Open-IG looks something like figure 2. In this case there are no violations and while the PlanetScreen class, which here is the pattern's client, appears to be complex the testing should not be affected by any real issues. At least not any issues caused by the command pattern's imposed relations.

![Figure 2. Class diagram showing classes in Open-IG related to the implemented command pattern.](image2)

**Conclusion**

In conclusion it can be said that a good way to measure the benefits of design patterns is through measuring its maintainability. A complete way to actually measure the maintainability has not been found, though one can of course get a subjective opinion through inspecting the code. The testability however, which is a large part of the maintainability, can be measured to an extent through counting the number of possible class interactions and possible self use interactions. These may be found in the class diagram of the implemented pattern.

In FreeCol the command pattern was extensively used and it can be said that not all of its effects were positive. Inspecting the maintainability through its different constituents analyzability,
changeability, stability and testability showed that the code was in some aspects a bit lacking. Primarily a possible self use interaction was found to be caused by the command pattern.

In Open-IG the pattern was implemented in a slightly different way causing the few deficiencies found to be a bit different, though still present. No self use interactions or class interactions were found however.

Overall the command pattern appears to have a positive effect. Though there were a few deficiencies in the different code bases one has to consider that faults would almost certainly also have been present given an alternate solution. Maybe the strongest argument against the pattern would be in the possible self use interaction found. This relation between classes was only found in one of the code bases though and can therefore not be assumed to be general to all implementations of this pattern.

Even if both code bases had shown identical properties it would have been bold to state something completely general regarding the usefulness of the command pattern. However, since the different implementations showed results pointing in both directions one could say the command pattern is not always good and neither always bad. It appears that it instead depends on variables that reside outside of the pattern template.

References


