How does the use of the Factory Method Pattern in FreeCol affect maintainability and understandability as measured with Coupling between object classes (CBO) and Number of children (NOC)?

Håkan Gudmundsson
Computer Engineering
Linköping University
Linköping, Sweden
Email: hakgu806@student.liu.se

***Abstract*—This paper will analyze what impact an implementation of the Factory Method design pattern has on a software's maintainability and understandability. The game FreeCol will be examined to determine the effect the Factory Method has and what purpose it serves. Maintainability is defined to have a close relation to coupling and understandability is defined to have a close relation to inheritance. The metrics CBO and NOC are proven to good when measuring these two software qualities. An implementation of the Factory Method pattern in FreeCol is examined and its pros and cons are discussed. The metrics and software qualities used are examined and discussed. The conclusion is that the Factory Method pattern have a positive effect on maintainability but a negative effect on understandability. However it seems like maintainability is a more sought after quality than understandability. So one might want to put this into consideration when deciding whether or not to use the pattern.**

I. Introduction

The concept of design patterns was brought up by the architect Christoffer Alexander, in 1977, as a solution to common problems in architectural designs. The basics of the concept was easy to understand and people began to experiment with implementing it in the architectural designs of computer programs. In 1987 Kent Beck and Ward Cunningham presented the results of such an experiment [6], but it would take a few more years before the concept really would gain popularity. In 1994, the book [1] was released, where the authors introduce the principles of design patterns in computer software and also offers a catalog of 23 design patterns that are still being used today.

Software is used in almost everything these days, in your phone, in cars, and even in your home. We depend a lot on software and therefore it's important that the quality is high. Chappell [2] highlights that there are three aspects when defining software quality, process quality, structural quality and functional quality. Process quality is about the developing process and has to do with meeting delivery dates and meeting budgets, structural quality is about the quality of the code itself and functional quality comes in to play when the software reaches its users and has a big focus on if the software met the specified requirements. This paper will look at an implementation of the Factory method in the game FreeCol[1] to analyze how it impacts maintainability and understandability. The analysis will consist of examining and referring to papers and studies related to software quality and software metrics. FreeCol is an open source, turn-based strategy game written in Java[2] similar to Civilization[3]. Since this paper is about analyzing and measuring software quality in code it will focus on the structural quality.

Implementing a design pattern isn't a guarantee that the quality of your software is going to increase. However as shown in [10], Zhang et. al. reaches the conclusion that there is some qualitative support for design patterns to provide a framework for maintenance. Another study, [9] also reports that 18 out of 23 of the patterns presented in [1] have been reported to have a positive effect on software maintainability.

"If we can't learn something, we won't understand it. If we can't understand something, we can't use it - at least not well enough to avoid creating a money pit. We can't maintain a system that we don't understand - at least not easily. And we can't make changes to our system if we can't understand how the system as a whole will work once the changes are made" [18]

The rest of the paper is organized as follows, section II covers background, section III presents the implementation of the factory method design pattern in FreeCol, in section IV the metrics results will be presented, in section V the implementation and general use of the factory design pattern is analyzed and in section VI the conclusion is presented.

II. Background

This section will cover the background and theory of the design pattern and software qualities and metrics that will be

[1] http://www.freecol.org/
[2] https://www.java.com/
[3] https://www.civilization.com/

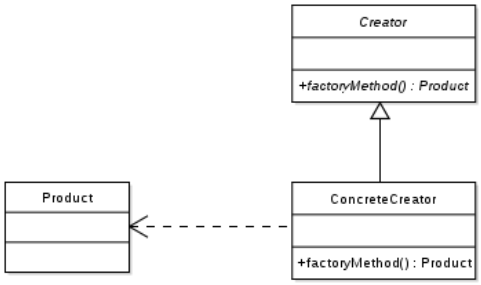

Fig. 1. UML of the Factory Method Design pattern

examined in this paper.

A. Factory Method pattern

Gamma et. al [1] introduces the intent of a Factory Method pattern as follows: "Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses". It is considered a creational pattern [1], meaning that it abstracts the instantiation process of objects.

As seen in fig. 1 the Factory Method encapsulates the knowledge of which *Product* subclass to create and moves this knowledge out to be decided by the *Concrete Creator*. This leads to lower coupling, since it decouples the client code in the superclass from the code that creates the object in the subclass. It also enforces the SOLID[4] principle, Dependency Inversion principle [12] in such a way that the dependencies of the client are solely to abstract classes and interfaces, and never to the concrete subclasses they are passed to.

Studies have show that, although the Factory Method pattern improves some areas of software quality, it doesn't only have positive effects [7] [8] [9]. Although the studies agree that the Factory Method pattern has a positive effect on maintainability, [7] finds that it also has a negative effect on understantability.

B. Software quality metrics

Software quality is a broad subject and may be understood different by different people. Therefore it is important to introduce some definitions of what view on software quality will be used in the analysis in this paper. As stated in the introduction, this paper will be focusing on the structural quality in its analysis. In the paper [2] Chappell some attributes of structural quality. These aspects are: code testability, code efficiency, code security, code understandability and code maintainability.

Maintainability is also bought up in the *CISQ's quality model* [4] as a desirable characteristic that is needed to provide value for a product. [4] lists some coding practices that is important have in mind when writing software with maintainability in mind. One of these practices listed is to look at how tightly coupled modules are. Coupling is, as described in [5], a measure of how closely connected two routines or modules are.

[4]https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

Understandability is described in the study [14] as a linear combination of abstraction, encapsulation, coupling, cohesion, polymorphism, complexity and design size. There's a lot of factors that can affect the complexity of software. In the study [15] the authors list some of the major principles behind Object-Oriented paradigm and their impact on software complexity, one of them being inheritance[5]. Inheritance makes it possible to create a class that is based on an other class and inherits methods and fields from that class. This might, if overused, lead to classes that are hard to understand [15], in other words the complexity of the classes are high.

1) CBO: A metric used for measuring coupling is *Coupling between object classes* (CBO) which was developed by Chidamber et. al. [13]. The CBO value of a class is a count of the number of other classes to which it is coupled. High CBO values means that the class is highly coupled and according to [16] and [13] would make it difficult to maintain.

2) NOC: As stated above, inheritance can lead to high complexity in classes. One metric used for measuring complexity, that also takes inheritance into consideration, is *Number of children* (NOC) which was developed by Chidamber et. al. [13]. This metric counts the number of classes that is a subclass (eg. inherits) from a chosen class. A high value of NOC means that the class have many subclasses. This might lead to the main class being hard to understand since in order to understand how the functionality of the main class you also must understand all of its subclasses [16].

3) Metric thresholds: Threshold values for CBO and NOC has been studied in [17]. The study concludes that the following values of CBO and NOC are acceptable and these threshold intervals will be used in the analysis in this paper.

TABLE I
THRESHOLD VALUES

Metric	Interval
CBO	0-8
NOC	0-6

When deciding the CBO and NOC values in this paper a program called *ckjm*[6] will be used. This program calculates the metrics proposed by Chidamber and Kemerer [13] by processing the bytecode of compiled Java files.

III. Implementation in FreeCol

In the game FreeCol an implementation of the Factory Method pattern is found in the package `net.sf.freecol.common.resources`. The pattern is used to decouple the use of the class `Resource` in the game. This is done by using the Factory Method `createResource` in the class `ResourceFactory` to create resource objects like `FAFileResource`, `SZAResource` etc. An UML diagram of how the classes are connected is show in fig. 2 and the code of the Factory Method is show in listing 1.

[5]https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
[6]http://www.spinellis.gr/sw/ckjm/

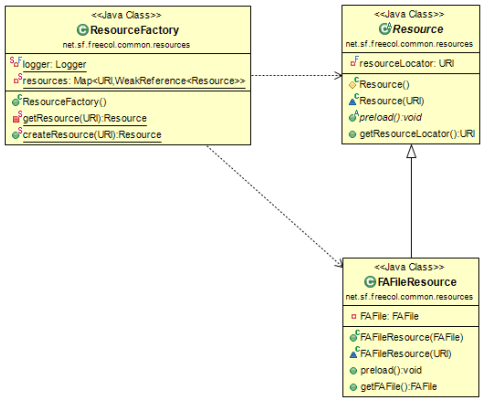

Fig. 2. UML of the Factory Method Design pattern as implemented in FreeCol

Listing 1. Implementation of the Factory Method pattern in FreeCol

```
public static Resource createResource(URI
    uri) {
  Resource r = getResource(uri);
  if (r == null) {
    try {
      if ("urn".equals(uri.getScheme()))
          {
        if (uri.getSchemeSpecificPart().
            startsWith(ColorResource.
            SCHEME)) {
          r = new ColorResource(uri);
        } else if (uri.
            getSchemeSpecificPart().
            startsWith(FontResource.SCHEME
            )) {
          r = new FontResource(uri);
        }
      } else if (uri.getPath().endsWith("
          .faf")) {
        r = new FAFileResource(uri);
      } else if (uri.getPath().endsWith("
          .sza")) {
        r = new SZAResource(uri);
      } else if (uri.getPath().endsWith("
          .ttf")) {
        r = new FontResource(uri);
      } else if (uri.getPath().endsWith("
          .wav")) {
        r = new AudioResource(uri);
      } else if (uri.getPath().endsWith("
          .ogg")) {
        if (uri.getPath().endsWith(".
            video.ogg")) {
          r = new VideoResource(uri);
        } else {
          r = new AudioResource(uri);
        }
      } else {
        r = new ImageResource(uri);
      }
      resources.put(uri, new
          WeakReference<Resource>(r));
    } catch (Exception e) {
      logger.log(Level.WARNING,
      "Failed to create resource with URI
          : " + uri, e);
    }
  }
  return r;
}
```

IV. Results

The results of the measurements of CBO and NOC are presented here. The metrics was taken from the classes `ResourceFactory` and `Resource`.

TABLE II
Results

Class	CBO	NOC
ResourceFactory	1	0
Resource	0	7

V. Discussion

In this section of the paper discussions are held regarding the implementation of the Factory Method pattern in FreeCol, the results of the measurements and the impact the use of the Factory Method pattern have on maintainability and understandability.

A. CBO and NOC

The classes that was included in this analysis was as stated above, `ResourceFactory` and `Resource`. The resource factory class was chosen because its the class containing the factory method. The resource class was chosen because it will give the most valuable information about maintainability and understandability using the CBO and NOC metrics, since measuring all the subclasses to the resource class will not say much unless the pattern is implemented inaccurately.

According to the results presented in table II and the threshold values in table I it is shown that both the classes are inside the acceptable interval when it comes to the CBO values. So according to [17] no design refinements are necessary.

When looking at the NOC values it appears that the class `Resource` has a slightly higher value than whats accepted according to the NOC threshold. This has to do with that the class is used in the Factory Method design pattern as an abstract class used to decouple the creation of the other resource classes. It means that the more different resource classes that exists and adapts the Factory Method pattern by extending the resource class, the higher the NOC value will be. According to [17] the class needs to be split to meet the required threshold value. One solution to lower the NOC value is to make even further abstractions of resources. Making subclasses to the subclasses of the resource class. This solution

would lower NOC value amongst classes but it would also introduce more abstractions which could lead to more complex software [15].

In the study [19] the researchers use the same metrics as in this paper (CBO and NOC) amongst other, to measure software quality. Their analysis is performed on the Java chart library *JFreeChart*[7] which is a software that is about the same size as FreeCol.

- FreeCol: 103800 lines of code
- JFreeChart (v.1.0.14): 11030 lines of code

The paper concludes that the metrics directly reflects the quality of the software. It also concludes that the quality of software decreases as it evolves.

B. Maintainability and understandability

Maintainability is defined in this paper with coupling and measured with CBO. The low value of CBO suggest that maintaining these classes would not be especially time consuming since low coupling means that the sensitivity to changes in other parts of the design is low [13] [16]. This is much thanks to the Factory Method pattern. If the pattern wasn't implemented, the responsibility of creating the concrete classes (the classes extending the resource class) wouldn't lie in subclasses of the resource class. One solution would be to simply let the resource class create hold the concrete classes. This would lead to much higher coupling since this class would be responsible and contain different information and behaviour depending on what resource it should represent. From a maintainability viewpoint this wouldn't be especially attractive since such a class would be very hard to make changes to due to the high coupling.

Understandability is defined in this paper as how many subclasses a certain class has and is measured with NOC. As stated above, the results in table II show that the NOC value from the resource class is 7, thus just outside the threshold values in table I. This is the result of the abstraction used in the Factory Method pattern. As all the different resources extends the abstract class resource, the more resources there are, the more subclasses will have to extend the resource class and thereby the NOC value will increase. If the solution above would be used the NOC value would be smaller, since none of the resource subclasses would exist. As a result, the class would, per definition, be easier to understand.

However, with this solution the class would break the Dependency Inversion principle, as the dependencies of the client (the game) would no longer be to the abstract class resource but instead to the concrete class resource. So the class would be referenced directly from the client, breaking the Inversion Dependency principle.

This paper have used metrics taken from [13]. There are other metrics that could have been used, for example, the metrics produced in the study [14]. However the metrics in [13] are focused on class level [20], and since this paper focused on analyzing an implementation of the Factory Method pattern in a larger software, this set of metrics was preferred.

[7]http://www.jfree.org/jfreechart/

C. Factory Method pattern

According to Gamma et. al. [1] design patterns makes a system less complex by letting you talk about it at a higher level of abstraction than if you where using program language. For example the phrase "Let's use a Factory Method here," which describes many steps of implementation, can be used instead of describing every step on its own.

In [21] the Factory Method patterns use in API design is studied. Instead of using software metrics as used in this paper, the study used participants who had considerable experience of the Factory Method pattern. The participants was given a task which they had to complete and then fill out a survey with their experiences. The study concludes that the pattern impairs usability, much because the participants found that the pattern was hard to understand.

It seems fair to assume that the Factory Method pattern have a negative effect on understandability and a positive effect on maintainability, based on the study made in this paper and other studies on the subject [7] [8] [9] [21]. So when choosing whether or not to use the Factory Method pattern based on maintainability and understandability one will have to decide which of the two qualities to prefer over the other. This, of course will differ from one situation to another. One way is to look at which quality is generally more attractive than the other. In [9] a mapping study is carried out on papers and studies and certain keywords for pattern effect on quality are counted. Maintainability and understandability are two of these keywords and its presented that maintainability is mentioned more often than understandability.

- Maintainability: 7 mentions in papers and studies
- Understandability: 5 mentions in papers and studies

The study also concludes that maintainability and understandability is one of the most commonly investigated quality attributes.

In the study [15] it's concluded that increased complexity does not necessarily have to decrease software quality. It's stated that: "Software complexity will only decrease software quality if it is poorly managed". When using a design pattern from the well known [1] it's hard to see that it would be considered as poorly managed complexity.

VI. Conclusion

There are many factors that have an impact on software quality. In this paper, maintainability and understandability have been studied and how the Factory Method pattern impact these two factors. The software quality maintainability is related to coupling, since a highly coupled software is hard to maintain. Understandability is related to inheritance, since to understand a class with many subclasses you have to understand every subclass. Popular metrics used to measure software quality on class level are CBO, for coupling and NOC, for inheritance. One study on software metrics suggests

that a value between 0-8 for CBO and a value between 0-6 for NOC is acceptable.

According to the metrics used, the Factory Method pattern have a positive effect on maintainability and a negative effect on understandability. Maintainability seems to be a more sought after quality than understandability, so one might put this into consideration when deciding whether or not to use the pattern.

In the game FreeCol the Factory Method pattern decrease coupling but also creates a lot of subclasses, to many according to the threshold values of NOC. However the value is just outside the threshold values and as stated, it's not necessarily the case that increased complexity decrease software quality.

REFERENCES

[1] Gamma, Erich. Design patterns: elements of reusable object-oriented software. Pearson Education India, 1995.

[2] Chappell, David. "The three aspects of software quality: Functional, structural, and process." (2013).

[3] Weyuker, Elaine J. "Evaluating software complexity measures." IEEE transactions on Software Engineering 14.9 (1988): 1357-1365.

[4] Soley, Richard Mark, and Bill Curtis. "The Consortium for IT Software Quality (CISQ)." International Conference on Software Quality. Springer Berlin Heidelberg, 2013.

[5] ISO, IEC. "IEEE, Systems and Software Engineering–Vocabulary." IEEE computer society, Piscataway, NJ (2010).

[6] Beck, Kent, and Ward Cunningham. "Using pattern languages for object-oriented programs." (1987).

[7] Khomh, Foutse, and Yann-Gael Gueheneuce. "Do design patterns impact software quality positively?." Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on. IEEE, 2008.

[8] Rajan, Hridesh, Steven M. Kautz, and Wayne Rowcliffe. "Concurrency by modularity: Design patterns, a case in point." ACM Sigplan Notices. Vol. 45. No. 10. ACM, 2010.

[9] Ampatzoglou, Apostolos, Sofia Charalampidou, and Ioannis Stamelos. "Research state of the art on GoF design patterns: A mapping study." Journal of Systems and Software 86.7 (2013): 1945-1964.

[10] Zhang, Cheng, and David Budgen. "What do we know about the effectiveness of software design patterns?." IEEE Transactions on Software Engineering 38.5 (2012): 1213-1231.

[11] Ellis, Brian, Jeffrey Stylos, and Brad Myers. "The factory pattern in API design: A usability evaluation." Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, 2007.

[12] Martin, Robert C. "The dependency inversion principle." C++ Report 8.6 (1996): 61-66.

[13] Chidamber, Shyam R., and Chris F. Kemerer. "A metrics suite for object oriented design." IEEE Transactions on software engineering 20.6 (1994): 476-493.

[14] Bansiya, Jagdish, and Carl G. Davis. "A hierarchical model for object-oriented design quality assessment." IEEE Transactions on software engineering 28.1 (2002): 4-17.

[15] Kalakota, Ravi, Sukumar Rathnam, and Andrew B. Whinston. "The role of complexity in object-oriented systems development." System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on. Vol. 4. IEEE, 1993.

[16] Shatnawi, Raed. "A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems." IEEE Transactions on software engineering 36.2 (2010): 216-225.

[17] Chandra, E., and P. Edith Linda. "Class break point determination using CK metrics thresholds." Global journal of computer science and technology 10.14 (2010).

[18] Nazir, Mohd, Raees A. Khan, and Khurram Mustafa. "A metrics based model for understandability quantification." arXiv preprint arXiv:1004.4463 (2010).

[19] Singh, Gagandeep. "Metrics for measuring the quality of object-oriented software." ACM SIGSOFT Software Engineering Notes 38.5 (2013): 1-5.

[20] El-Wakil, Mohamed, et al. "Object-oriented design quality models a survey and comparison." 2nd International Conference on Informatics and Systems (INFOS 2004). 2004.

[21] Ellis, Brian, Jeffrey Stylos, and Brad Myers. "The factory pattern in API design: A usability evaluation." Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, 2007.

IMPROVEMENTS AFTER SEMINAR 6

I hadn't written much of the paper on smeinar 6. My title was unclear and i hadn't decided what software qualities to measure and what metrics to use. The feedback i received on the seminar mostly consisted of tips on how to get started. I got tips on how to think when choosing the metrics to used and Joakim sent me an email with some good papers to get me started.