Factory method - Increasing the reusability at the cost of understandability

The author
Linkping University
Linkping, Sweden
Email: liuid123@student.liu.se

Abstract—This paper describes how Bansiya and Davis’s well cited software quality suit contradicts previous results regarding the effects the factory method design pattern has on understandability. That is, the metric suite has an inbuilt bias towards increasing the understandability when one uses simple designs of the factory method in larger designs. This contradicts the previous results that shows that the understandability decreases when one uses the factory method design pattern. Moreover, it also shows that the metric can agree with previous results regarding the effects the same pattern has on reusability.

I. INTRODUCTION

The effects of using design patterns for object oriented programming has been studied quite thoroughly. For example, Ampatzoglou, Charalampidou and Stamelos did a mapping study, a type of systematic literature study, about the effects of design patterns on software quality in 2013 that used 118 papers as their data [1]. Another interesting area of research is that of software metric. That is, a metric that measures the overall, or parts of, the quality of the code. There are many metrics, based on different principles, of this sort. Given all the work put into these metrics, it’s not far fetched that one evaluate the effects of a design pattern by using one or more of them. However, only 6 of the 118 papers that Ampatzoglou, Charalampidou and Stamelos included in their paper used software metrics to evaluate the effects of the design pattern [1]. They state that using metrics isn’t better than using expert knowledge in controlled experiments. However, this low amount of studies based on metrics leaves one question open. Can one expect different results regarding the effects of design pattern on software quality compared to that of the controlled experiments? Moreover, is there an inbuilt agreement with the effects of the factory method design pattern on software quality?

To do a general study of all the design patterns, all aspects of software qualities and all the software metrics that exists is not possible within the limits of this study. Therefore, this paper is restricted to study the factory method design pattern and its effect on the reusability and understandability aspects of software quality. Moreover, the paper will be restricted to the metrics defined by Bansiya and Davis [2]. As a result, this paper aims to answer the following questions:

1) Can reusability, according to Bansiya and Davis’s metric suite, be increased by refactoring so that the factory method design pattern is used?
2) If so, is it at the cost of decreasing understandability? Again according to Bansiya and Davis’s metric suite.
3) Do the metric in question have an inbuilt agreement with the effects of the factory method design pattern on software quality?

II. THE FACTORY DESIGN PATTERN

A. The pattern

Gamman et al (commonly refereed to as the gang of four or GoF for short) defined the factory method design pattern as one of their creational patterns that in their book Design Patterns - Elements of Reusable Object-Oriented Software [3]. The purpose of the pattern is to abstract the creation of an object for which the caller only wants or needs to know that it derives from a specific class. i.e. the caller does not want or need to know which derived class it gets. Therefore, the caller delegates the creation to a factory. Thereby, abstracting the creational process.

The pattern consists of a Product, one or more ConcreteProducts, a Creator and zero or more ConcreteCreators[3]. With this naming the Product is a base (abstract) class or interface and the ConcreteProducts are the derived classes. These are the classes that might be instantiated but the caller only wants to know which one but only that it is a Product. The Creator is the base class or interface for the classes that defines the factory method and the ConcreteCreators are its derived classes (if any exists).

B. Observed effects

Throughout the study of design patterns it has been observed that they can have both positive and negative effects on the software quality [1]. Interestingly, these are not necessarily mutex in each case. If one divides software quality into different sub-properties\(^1\), then one will find that a design pattern can affect one of the sub-properties of software quality positively while another is affected negatively.

\(^1\)There exists many categorisations of software quality into sub-properties. One of the more known is the original ISO 9126-3 standard and the later ISO/IEC 25000:2014. However, there are many more categorisations and a significant amount of them are based on the previously mentioned ISO standard(s).
Khomh and Gueneucue carried out a quantitative study of design patterns to determine the effects on software quality. In their study they found that the factory method had a positive effect on the properties that they called expandability and reusability. However, they also got the result that the very same pattern had a negative effect on the understandability of the software [4].

III. SOFTWARE METRICS

Bansiya and Davis have defined a set of metrics that can be weighted together to give an value for the different properties of software quality that they have defined[2]. These properties are reusability, flexibility, understandability, functionality, extendability and effectiveness. Out of these only reusability and understandability is of interest for this study. In their article Bansiya and Davis connects each the software properties to object oriented design properties that affects the specific software property. Thereafter, the design properties are connected to a metric and their normalised values are weighted together for each software quality property as a compound metric. A summary for how the reusability and understandability quality properties is broken down to metrics [2] can be found in table I. Unfortunately, the description of the metrics are a bit vague in the original paper and some references are simply not available. Moreover, no documents aiming to explain these metrics in details was found. Fortunately, Keeffe and Cinnide included a detailed interpretation of the metrics in their paper [5]. The following sections will describe the relevant metrics:

A. Average number of ancestors

The average number of ancestors (ANA) is a metric that aims to measure the abstraction software property. It is calculated as follows [2]:

1) For all classes calculate the number of ancestors at any level. For example, if class A inherit class B which in turn inherit class C, then A gets the value 2.
2) Calculate the average of the values from step 1. In the example above, this becomes \( \frac{2+1+0}{3} = 1 \).

B. Data access metric

The data access metric (DAM) aims to calculate how well encapsulated the software is. This is calculated by summing the number of private and protected attributes in a class and divide it by the total number of attributes in the class [2]. This metric is class centred and can be combined to cover the whole design by taking the average value of the DAM value for all classes with at least one attribute in the design [5]. The averaged metric will be called averaged data access metric (aDAM) in this paper to be clear.

C. Direct class coupling

The direct class coupling (DCC) metric estimates how coupled a class is. This is done by counting the number of classes (excluding itself) it is connected to through attribute declaration and message passing in methods (parameters) [2]. Obvious, this is a class oriented metric which can be translated to a design level metric by taking the average of all the classes [5]. Moreover, the original description of the metric allows for interpreting that the library classes should be counted to this. However, Keeffe and Cinnide excludes these library functions from the metric [5]. The resulting design metric will be called averaged direct class coupling (aDCC) in this paper for clarity.

D. Cohesion among methods in class

Cohesion among methods in class (CAM) is a class centred metric that is calculated as follows [2]:

1) Count all the parameter types to all the methods in the class.
2) For each method, divide the number of parameter types for the method with the value from step one.
3) Take the average of the calculated value for all the methods in the class.

To make this metric a design metric instead of a class centred metric the average value for all the classes is used [5]. Moreover, any implicit parameter to the own class is ignored. In addition, a class without parameters will get the value 1. The resulting design metric will be called averaged cohesion among methods in class (aCAM) in this paper.

E. Number of polymorphic methods

The number of polymorphic methods (NOP) is counted as the number of methods in a class that can express polymorphic behaviour [2]. Keeffe and Cinnide defines "polymorphic behaviour" in more generic way than the original “Such methods in C++ are marked as virtual” [2], as a method that is overridden by any descendent class. This allows for calculating the metric for languages that does not use such constructs, for example Java. Moreover, they make it a design centred metric by taking the sum of NOP for all classes (henceforth referred to as sum of number of polymorphic methods, sNOP).

F. Number of methods

The number of methods (NOM) is simply calculated as the number of methods in a class [2]. From this class centred metric a design centred metric can be gained by taking the sum over all the classes in the design [5]. The resulting metric will be called sum of number of methods (sNOM) in this paper.

G. Design size in classes

The design size in classes (DSC) metric is calculated by counting all the classes in the design [2].

H. Class interface size

The class interface size (CIS) metric is calculated by counting all the public methods in a class [2]. To extract a design centred metric from this class centred metric, one takes the average of the CIS value for all classes [5]. This metric will be called average class interface size (aCIS) in this paper.
IV. Method

To be able to answer the research questions presented in the introduction, the study is divided into two parts. First is a case study that aims to answer the first two questions regarding if the effects of implementing the factory method design pattern according to Bansiya and Davis’s metric suite can agree with the previous results regarding understandability and reusability. The second part is a formal transformation of case when the factory method can be applied. This will then be used as a basis for determine if the metric suite agrees with the observed effects of the factory method design pattern. The following to sections describes the methodology for these two parts.

A. Case study FreeCol

To evaluate if the factory method increases the reusability and decreases the understandability according to the metric suite presented by Bansiya and Davis [2] a case study will be done. The study will be conducted using a subset of the code for the game FreeCol\(^2\), the package net.sf.freecol.common.option. When the metrics are calculated, the values are normalised with respect to the version before the factory method was implemented. This is the same step taken by Bansiya and Davis in the original paper [2]. Finally, the metrics are weighted together according to the formulas in table I. Note that a higher value is better for both understandability and reusability.

B. Inbuilt metric bias

When evaluating if Bansiya and Davis’s metric suite [2] agrees with the observed effects a formal transformation will be done. This means that there will be a transformation model for how the design changes when refactoring so that it implements the factory method design pattern. Based on this model, parametrised metric values will be calculated and compared. The parametrised metric values can then be compared to see if and under which conditions that Bansiya and Davis’s metric suite agrees with the observed effects of the factory method.

V. Results

A. Case study FreeCol

1) Changes in the code: In the net.sf.freecol.common.option package, there’s a method called readOption method in the abstractOption class. The method is one of all the methods spread over the whole implementation of the game that loads data from xml files (for example loading scenarios). However, this method contains a part shown in listing 1 that is a common structure for a factory method. That is, the method itself isn’t concerned with which derived class it creates just that its super class is abstractOption. Moreover, having this functionality as a part of a method which responsibility is something else yields a lower reusability since this part cannot be called from another place.

Refactoring this part of the code by using the factory method design pattern resulted in a new class called OptionFactory. This class consists of a static map and a public static method in listing 2. This allows changing the readOption method so that it no longer is responsible

<table>
<thead>
<tr>
<th>Quality properties</th>
<th>Relevant design properties</th>
<th>Weighted metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understandability</td>
<td>Abstraction, encapsulation, coupling, cohesion, polymorphism, complexity and design size</td>
<td>-0.33<em>ANA+0.33</em>aDAM-0.33<em>aDCC+0.33</em>aCAM-0.33<em>sNOP-0.33</em>sNOM-0.33*DSC</td>
</tr>
<tr>
<td>Reusability</td>
<td>Coupling, cohesion, messaging and design size</td>
<td>-0.25<em>aDCC+0.25</em>aCAM+0.5<em>aCIS+0.5</em>DSC</td>
</tr>
</tbody>
</table>

\( ^2 \)FreeCol is a free and open source clone of Sid Meier’s Colonization. The code can be acquired from their website http://www.freecol.org
**Figure 2.** The new `OptionFactory` class which is responsible for creating options.

```java
final class OptionFactory {
    private static Map<String, Constructor> constructorMap = new HashMap<>();
    static {
        // Populating the map
        [...]
    }
    static AbstractOption createOption(
        String tag, Specification spec)
    try {
        return (AbstractOption) constructorMap.get(tag).newInstance(spec);
    } catch (InstantiationException | InvocationTargetException e) {
        logger.finest(
            "Unable to create option of type \" + tag + \"\" \" + tag);
        return null;
    } catch (IllegalAccessException e) {
        logger.finest("Parsing of option, type \" + tag + \" is not implemented yet\");
        return null;
    }
}
```

**Figure 3.** The new version of the `readOption` method that is using the newly created `OptionFactory`.

```java
protected AbstractOption readOption(
    XMLStreamReader in)
throws XMLStreamException {
    [...]
    AbstractOption option = null;
    if (ACTION_TAG.equals((tag))) {
        logger.finest("Skipping \"action\" + getAttribute(in, ID_ATTRIBUTE_TAG, \"\"));
        in.nextTag();
    } else {
        option = OptionFactory.createOption(tag, spec);
    }
    return option;
}
```

for knowing which tags that corresponds to the different file types as shown in listing 3. The gain from this is that the `AbstractOption` is decoupled from the classes that derives from it and that the code can now be reused to create options in other places as well.

2) **Software quality effects:** The changes in the code mentioned in the previous section had the effect on software quality in that the reusability increased by 6% and that the understandability increased by 8%. All the metric values and their normalisation is shown in table II.

**B. Inbuilt metric bias**

The factory design pattern that is used for the transformation model is the simplest for of the textbook implementation. That is, the factory will only consist of one class that contains one public method and no private or protected methods. The new metric values after the transformation has been applied as well as the normalised values are shown in table III. A short description of how these values were acquired follows here:

- **ANA:** The factory class does not have any ancestors and therefore only modifies this metric by increasing the design size by 1.
- **aADAM:** The factory class does not have any attributes at all (only a public method). Therefore, this metric is only changed due to the larger design size.
- **aDCC:** The only possibility to make a general transformation for this metric is to introduce a new variable. This variable, called `aP` in this paper, is the number of different non-library classes that are parameters to the method in the factory pattern. By using this variable one can get the aDCC after the transformation.
- **aCAM:** There is only one method in the factory implementation and therefore the CAM for that class is always one.
- **sNOM:** The factory method does not inherit from any other class and no classes inherit from it. Therefore, this metric value change only because the design size change.
- **DSC:** The design size increases with one since one new class is introduced.
- **aCIS:** The factory method has one public method which affects this metric.

Calculating the changes in understandability, abbreviated to `u`, from the normalised values in table III is done as follows (according to the formula for understandability in table I):

\[
    u = -0.33 \frac{DSC}{DSC + 1} + 0.33 \frac{DSC}{DSC + 1} - 0.33 \frac{aP}{aDCC + aP} + 0.33 \frac{aCAM \cdot aP}{aDCC + aP} - 0.33 * 1 + 0.33 \frac{sNOM + 1}{sNOM} - 0.33 \frac{DSC + 1}{DSC} \]

\[
    = -0.33 \frac{DSC}{DSC + 1} + 0.33 \frac{DSC}{DSC + 1} - 0.33 \frac{DSC + \frac{aP}{aDCC}}{DSC + 1} + 0.33 \frac{DSC + \frac{aCAM}{aDCC}}{DSC + 1} - 0.33 + 0.33 \frac{0.33}{sNOM} - 0.33 - 0.33 \frac{DSC}{DSC} \]

The interesting part is when this value is smaller than the normalised value before the transformation (the understandability for the base is always -1 with this metric):
The value for reusability, abbreviated to \( r \), can be calculated similarly as the understandability was calculated (based on the formula for reusability presented in table I):

\[
r = -0.25 \frac{aDCC \times DSC + aP}{DSC + 1} + 0.25 \frac{aCAM \times DSC + 1}{aCAM} + 0.5 \frac{aCIS \times DSC + 1}{aCIS} + 0.5 \frac{DSC + \frac{1}{aCAM}}{DSC + 1} + 0.25 \frac{DSC + \frac{1}{aCAM}}{DSC + 1} + 0.5 \frac{aCIS - 1}{DSC + 1} + 0.5 \frac{1}{DSC} - 1
\]

\[
\equiv -0.25 \frac{DSC + \frac{aP}{aDCC}}{DSC + 1} + 0.25 \frac{DSC + \frac{1}{aCAM}}{DSC + 1} + 0.5 \frac{aCIS - 1}{DSC + 1} + 0.5 \frac{1}{DSC} - 1
\]

Again the interesting part is when the reusability is larger after the transformation than before the transformation (the reusability for the base is always 1):

\[
r \geq 1
\]

\[
\equiv 1 - 0.25 \frac{DSC + \frac{aP}{aDCC}}{DSC + 1} + 0.25 \frac{DSC + \frac{1}{aCAM}}{DSC + 1} + 0.5 \frac{aCIS - 1}{DSC + 1} + 0.5 \frac{1}{DSC} - 1
\]

\[
\equiv -0.25DSC - 0.25 \frac{aP}{aDCC} + 0.25DSC + 0.25 \frac{1}{aCAM} + 0.5 \frac{aCIS}{aDCC} + 0.5DSC + 0.5 \frac{1}{aCAM} - 2
\]

\[
\equiv - DSC - \frac{aP}{aDCC} + DSC + \frac{1}{aCAM} + \frac{1}{aCIS} - 2
\]

\[
\equiv 1 + 2DSC + \frac{1}{aCAM} + \frac{DSC + 1}{sNOM}
\]

\[
\leq \frac{aP}{aDCC} + \frac{DSC}{DSC + 1}
\]

The case study shows that the understandability and the reusability both increased when the code was refactored so that it uses the factory method design pattern. This is enough to answer the first question regarding if one can increase the reusability according to Bansiya and Davis’s metric suite by using the factory method. Unfortunately, the results regarding understandability is not enough to say anything regarding the second question since it is not a positive result and not general enough to decisively answer the question with a no.

As one can see, neither (1) nor (2) is always true. Instead, they express a constraint for when the effects of the factory method design pattern agrees with the previous results.

VI. DISCUSSION

A. Case study FreeCol

The case study shows that the understandability and the reusability both increased when the code was refactored so that it uses the factory method design pattern. Unfortunately, this is

### Table III

<table>
<thead>
<tr>
<th>Metric</th>
<th>After transformation</th>
<th>Normalised</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANA</td>
<td>ANA + DSC</td>
<td></td>
</tr>
<tr>
<td>aDAM</td>
<td>aDAM + DSC</td>
<td></td>
</tr>
<tr>
<td>aDCC</td>
<td>aDCC + DSC + aP</td>
<td></td>
</tr>
<tr>
<td>aCAM</td>
<td>aCAM + DSC + 1</td>
<td></td>
</tr>
<tr>
<td>sNOP</td>
<td>sNOP + 1</td>
<td></td>
</tr>
<tr>
<td>sNOM</td>
<td>sNOM + 1</td>
<td></td>
</tr>
<tr>
<td>DSC</td>
<td>DSC + 1</td>
<td></td>
</tr>
<tr>
<td>aCIS</td>
<td>aCIS + DSC + 1</td>
<td></td>
</tr>
</tbody>
</table>

Again the interesting part is when the reusability is larger after the transformation than before the transformation (the reusability for the base is always 1):

\[
r \geq 1
\]

\[
\equiv 1 - 0.25 \frac{DSC + \frac{aP}{aDCC}}{DSC + 1} + 0.25 \frac{DSC + \frac{1}{aCAM}}{DSC + 1} + 0.5 \frac{aCIS - 1}{DSC + 1} + 0.5 \frac{1}{DSC} - 1
\]

\[
\equiv -0.25DSC - 0.25 \frac{aP}{aDCC} + 0.25DSC + 0.25 \frac{1}{aCAM} + 0.5 \frac{aCIS}{aDCC} + 0.5DSC + 0.5 \frac{1}{aCAM} - 2
\]

\[
\equiv - DSC - \frac{aP}{aDCC} + DSC + \frac{1}{aCAM} + \frac{1}{aCIS} - 2
\]

\[
\equiv 1 + 2DSC + \frac{1}{aCAM} + \frac{DSC + 1}{sNOM}
\]

\[
\leq \frac{aP}{aDCC} + \frac{DSC}{DSC + 1}
\]
not something one can easily verify and will therefore be left as a potential unexplored explanation. Secondly, there is the possibility that Bansiya and Davis’s metric suite fails to measure the software qualities correctly. The final explanation that will be covered is the possibility of miscalculations when calculating the metric values. However, the risk of this happening has been reduced by recalculating the metric values at a later point in time.

The second explanation is just like the first hard to verify. However, there are some interesting points here. Firstly, understandability is usually defined in terms of how easy software is understood. Naturally, this allows for subjective measurement since, for example, experience potentially could influence what is easy to understand. Therefore, it is not certain that Bansiya and Davis’s metric suite for understandability takes the same aspects of the software code into consideration than what was done in the previous studies regarding understandability. This would not be a strict flaw in the metric, only a difference. Unfortunately, there are issues with the metric as well. For example, the DDC metric aims to measure how coupled a class is. However, it does not take static classes into consideration. Similarly, it does not count the classes used inside the method bodies since it is a metric for design level and not implementation level.

### B. Inbuilt metric bias

The results from this part of the study are two constraints that has to be fulfilled for the metric suite to agree with the previous studies. In a sense, this means that the metric suite does not agree with the previous results since it is not unconditional. However, by looking closer at the constraints in the result one can gain more information. As seen in (1), the understandability constraint is dependent on \( \frac{1}{\text{aCAM}}, \frac{\text{DSC}}{\text{sNOM}}, \frac{\text{aP}}{\text{DCC}} \) and \( \frac{1}{\text{aCIS}} \). We know that \( \frac{1}{\text{sNOM}} \leq 1 \) which allows for the simplification of the criteria to:

\[
2 \text{DSC} + \frac{1}{\text{aCAM}} + \frac{\text{DSC} + 1}{\text{sNOM}} \leq \frac{\text{aP}}{\text{DCC}}
\]

Moreover, it’s quite reasonable that the following holds for larger design sizes:

\[
2 \text{DSC} \geq \frac{\text{aP}}{\text{DCC}}
\]

This is because the left hand side grows linearly to the number of classes in the design while the right hand side grows linearly with the number of different non-library classes that the factory class is connected to through the method parameters. This cannot be expected to grow linearly with the design size but rather to reach a quite stable value since the factory is not supposed to handle construction of all classes in the design.

Based on the above, it follows that the constraint for understandability is false under all circumstances when the design size is large and the assumption done above is valid.

Therefore, it can be expected that the metric will yield that the understandability increases for larger design sizes.

The constraint for reusability, (2), depends on \( \frac{1}{\text{aCAM}}, \frac{\text{DSC}}{\text{sNOM}}, \frac{\text{aP}}{\text{DCC}} \). As reasoned above \( \frac{\text{aP}}{\text{DCC}} \) is likely to approach a constant value as the design size grows. Moreover, \( \frac{\text{DSC}}{\text{sNOM}} \) will approach 0 as the design size grows. Unfortunately, \( \frac{1}{\text{aCAM}} \) is also likely to be stabilising as the design size grows. This means that when the design size grows the constraint will approach:

\[
\frac{1}{\text{aCAM}} + \frac{2}{\text{aCIS}} \geq \frac{\text{aP}}{\text{DCC}}
\]

All of these metrics are specific for each project (aCAM depends on the parameter types in the classes and aCIS depends on how many public methods the classes have). Hence, there is no way to conclude for a general case if this constraint will evaluate to true or false.

Finally, one should note that the two constraints are based on implementing a very basic version of the factory method design pattern. Hence, any conclusion based on this assumption must be regarding the most simple case. To be able to draw a general conclusion, the constraint must be generalised to cover the general implementation of the factory method design pattern as well.

## VII. Conclusion

The conclusions that can be drawn from this study are (in the order of the research questions):

1) The factory method design pattern can increase the reusability according to Bansiya and Davis’s metric suite.

2) Based on the results in this study it is not possible to draw a conclusion about if the understandability can decrease by implementing the factory method design pattern.

3) The metric does not have an absolute inbuilt agreement with the previous results. That being said, the metric will disagree with the previous results regarding understandability as long as \( 2 \text{DSC} \geq \frac{\text{aP}}{\text{DCC}} \) and the implemented factory method has a simple class design, as used in this study. This means that the metric is unlikely to agree with the previous results as long as the design size is larger and the factory method has a simple class design.

## VIII. Further Studies

There are some interesting questions that have not been answered in this study left open. One of them is validating that Bansiya and Davis’s metric suite manage to measure understandability in a good manner independent on the subjectivity of understandability. This requires a wider study where designs are evaluated by practitioners with different background, experience and knowledge. Another interesting study to expand the constraints from this study to be more general and to generate more constraints for other design patterns as well. Given such criterion one can validate Bansiya and Davis’s metric suite by comparing them with previous
results regarding the effects design patterns has on software quality. Moreover, if the metric suite shows to be valid, then these constraints can be used as a basis for automatically give suggestions if a design pattern could be used to increase the software quality to the developers and designers.

REFERENCES


