OBJECT-ORIENTED LANGUAGES: DEFECTS AND SHORTFALLS

Aiman Darwiche
Graduate School of Computer and Information Sciences, Nova Southeastern University,
Fort Lauderdale, Florida, USA

ABSTRACT

The popularity of object-oriented languages requires serious measures to look into the shortfalls of the languages and study the defects that might result from design flaws. The vast amount of software developed using this paradigm requires certain methods and techniques to keep the quality high by finding the defects and shortfalls. The paper will discuss some of the design flaws and defects and look at methods of detecting them and we will look at object-oriented shortfalls namely the crosscutting concerns and the techniques to find and correct those using Aspects and Aspect-Oriented methods and modals.

Keywords: AOP, Aspects, Crosscutting, Defects, Metrics, Object-Oriented, OOP.

1. INTRODUCTION

The widespread use of object-oriented analysis and design (OOAD) as a software engineering approach makes it important to examine the defects in the design of the applications utilizing OOAD and study the shortfalls of the object-oriented languages as well. The complexity level of the object-oriented design such as the high depth level of the inheritance tree increases the possibility of falling into defects[1]. Another cause of defects is the existence of crosscutting concerns, which are scattered and tangled code implementations across an entire application[2].

The popularity of object-oriented design and development requires measures to minimize the number of flaws in applications that can lead to defected software, which in turn will cause major problems and incur unnecessary costs. Many of the flaws of object-oriented designs (OOD) can be simply avoided by sticking to standards and simplifying the designs. In addition, experience and knowledge help reduce the defects. Nevertheless, it is very important to come up with methods that find defects in the designs to keep delivering high quality software.
Researchers came up with many metrics-based techniques to detect the defects. One technique used the covariance matrix and info-gain for attribute selection[3]. Some used the Chidamber & Kemerer (CK) object-oriented metrics suite[4], [5], while others used dependency-oriented complexity metrics [6].

Crosscutting concerns resulted in the emergence of Aspect-Oriented Programming (AOP) as a solution to the shortfalls of Object-Oriented Programming (OOP). AOP has been generating lots of attention in the software engineering community on both the academic and business levels. AOP has improved the quality of OOP by eliminating crosscutting concerns such as in logging, concurrency, persistence, authentication, and transaction management[7].

The rest of this survey paper consists of four sections. The second section will present the different defects and shortfalls in OOP. The third section will discuss different methods and models to identify and detect these defects. The paper will propose suggestions for future work in the fourth section, and then closes with the conclusion.

2. DEFECTS

Defects in software, most commonly known as bugs, are of several types. They can simply be an error in the program that does not cost many efforts to correct. They can also be implementations that have not addressed the design requirements properly or faulty calculations due to the misunderstanding of these requirements. The other type, which is the topic of this paper, are defects that are the result of the complexity of the OOP paradigm or the result of the scattered code known as crosscutting concerns.

2.1 OOP Defects

OOP is great at solving complex software problems but at a price. The paradigm itself holds the seeds of defective software due to complexity and misuse.

Taylor and Haddad [1] came up with a list of weaknesses that usually results in flaws and defects. These faults relate to the essence of OOP [1].

A major weakness is inheritance. This is one of the most used features in OOP, but it is also the main cause of problems [1]. Overuse of inheritance results in tightly coupled classes, which increases the cost of maintenance and reduces the legibility and understandability of the code. In addition, the level of the inheritance tree confuses the location of the code and complicates the code learnability [1]. Besides, inheritance can lead to duplication in code, by creating similar methods in different classes [1]. As an example, Taylor and Haddad [1] suggested that two methods in two different classes may only have a 10% change in their functionalities. Programmers cannot just override 10% of a method. If the programmers override the whole method, the result is 90% of duplicate code. This code duplication will make maintenance or bug fixing very difficult. The other choice will be modifying the original method by moving it up to the main class. This will require the main class to know about its children thus ruining the cleanliness of the class hierarchy. Another complication that may arise from modifications is the inability to restructure class hierarchy tree. In addition, modifications of the subclasses can result in subclass that is not a truly inherited one thus violating the polymorphism feature of OOP [1].

Encapsulation is a very powerful feature in OOP. It prevents unwanted and unintentional changes, but in a complex system, it may hinder legitimate parts of the application to do changes that they are entitled to do, therefore, reducing the strength of this feature and resulting in complicated workarounds to achieve the intended goal [1].

The concept of classes is a core one in OOP. However, according to Taylor and Haddad [1], classes are not suitable for dynamic domains. In a highly changing environment, classes are very rigid in responding to changes, for example the US tax code, which changes yearly [1].
For weaknesses with lesser impact, Taylor and Haddad [1] cite improper training and awkward language features as reasons for defective OO software.

2.2 Crosscutting concerns
As mentioned earlier, crosscutting concerns are scattered and tangled code implementations across an entire application [2]. Eaddy et al. [2] proposed that since crosscutting concerns consistently change, they are harder to implement and update as they are spread in multiple and unrelated positions. In addition, developers may have hard time understanding the reason behind scattering the code, and will have a harder time mentally untangling the distributed code from the code that relates to other concerns [2]. Accordingly, the presence of crosscutting concerns increases complexity thus leading to increased defects [2].

3. DETECTION
Even with the utmost efforts that developers and programmers exert to produce bug-free and reliable software application, defects slip and show up after deliveries of these programs. The best approach to development is defect avoidance - being a defensive programmer. However, if all these efforts fail, defect detection comes to the rescue [7].

3.1 OO Detection
Researchers had put a great effort in defect detection. Most of the work revolved around metrics. They used existing metrics, extended the existing sets, or came up with new metrics either very new or variations of the current ones.

Mishra and Shukla [8] aimed at building a better prediction model to predict defects by analyzing the impact of attribute selection techniques on class performance [8]. For the full set of attributes, see Table 1. They used two metrics. The first one in the info-gain, which is a method that ranks the attributes in relation to their gain score from the most informative to the least. The second metric is the covariance matrix, which is a method that calculates the covariance for all attributes [8].

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBO</td>
<td>Coupling between objects</td>
</tr>
<tr>
<td>DEPTH</td>
<td>Depth of inheritance</td>
</tr>
<tr>
<td>LCOM</td>
<td>Lack of cohesion</td>
</tr>
<tr>
<td>NOC</td>
<td>Number of children</td>
</tr>
<tr>
<td>DOC</td>
<td>Dependence on an descendent</td>
</tr>
<tr>
<td>FAN_IN</td>
<td>Count of calls of higher modules</td>
</tr>
<tr>
<td>RFC</td>
<td>Response for a class</td>
</tr>
<tr>
<td>WMC</td>
<td>Weight method per class</td>
</tr>
</tbody>
</table>

Their model showed some positive results especially with careful selection of attributes, which can affect the prediction negatively if poorly selected [8].

Abuasad and Alsmadi [9] looked at creating a fault prediction model also, using a bug prediction approach with software historical data. They used a set of coupling metrics namely: coupling between object (CBO), response for class (RFC), Message Passing Coupling (MPC), Data Abstraction Coupling (DAC, DAC1), Information-flow Based Coupling (ICP), Number of Children (NOC), Number of Ancestor Classes (NAC), Number of Descendent Classes (NDC), ACAIC, ACMIC, AMMIC, DCAEC, DCMEC, and DMMEC [9]. They outlined these steps to build their model:
1. Calculate coupling metrics.
2. Collect fault data.
3. Classify classes as faulty or not faulty.
4. Perform the Principal Component (PCA) Analysis.
5. Build the prediction fault model using machine-learning techniques
6. Calculate the accuracy of the model based on confusion matrix.

The researchers concluded that using the coupling metrics could be a good indicator for software and maintainability. Their model can help in early evaluation of problems to lower coupling and consequently improve quality [9].

Herraiz el al. [4] looked at the CK metrics suite and tried to come up with a better understanding of the metrics and enhance the way to use them to get better results. They examined the weighted methods per class (WMC), Depth of inheritance tree (DIT), Number of children (NOC), Coupling between object classes (CBO), Request for a class (RFC), Lack of cohesion in methods (LCOM). Although they did not work on defect detection, but improving the usage of the metrics helped other researchers identify defects more effectively [4].

Danphitsanuphan and Suwantada [10] suggested an approach to detect code smells. Code smells are simply bad source code inside an application that results in difficulty comprehending the software application and hinders improvement of that application. There are 22 types of code smells but only some can be measured by software metrics such as The Large Class, Long Method, Long Parameter, and Lazy Class. They used several metrics to measure the defects at the class and method levels. The class level methods are Number of Methods Class, Lines of Code Class, Depth of Inheritance Tree Class, Weighted Methods per Class, Number of Attributes, NCS Number of Children, and Lack of Cohesion of Methods. The method level metrics are Number of Parameters Method, Method Lines of Code Method, and McCabe Cyclomatic Complexity Method [10]. Using these metrics, they implemented the Bad Smell Detection Tool (BSDT) as a plug-in for Eclipse. BSDT was capable of detecting bad smell in various Java code with high accuracy compared to testing [10].

Taylor and Haddad [1] elaborated the flaws in the OO paradigm, but did not offer any model or tool. They mainly suggested some recommendations to improve OOAD. They recommended the following:

1. Design is a continuous process – meaning the ability to change the design to accommodate future changes without breaking the software.
2. Lesser use of deeper inheritance hierarchy – the flatter the design the better.
3. Proper training in both the procedural and OO paradigms.
4. Suitable teaching in code maintenance skills.

Mishra and Shukla [3] built another prediction model using similar attributes and metrics they used previously but incorporated Fuzzy Logic and genetic algorithms to predict defects. Their experiments showed its effectiveness, which can have a positive impact by alleviating the miss detection risks of fault prone development in the early stages [3].

3.2 Crosscutting Detection

Crosscutting concerns had the focus of many researchers interested in improving OO software quality. Research concentrated on tools and techniques to identify these concerns and remove them resulting in the birth of an extension to OOP languages called Aspect-Oriented programming (AOP). Crosscutting concerns are known as Aspects in the AOP world.
Sardinha et al. [11] worked on a tool they called EA-Tracer. The purpose of the tool is to automate the identification process of the aspects at an early stage, and trace the links of these early aspects to code aspects, hence the name the Early Aspect (EA) Tracer [11]. The EA-Tracer process has three steps. The first one is to find the textual early aspects in the requirement documents using part of the tool called the EA-Miner. The second step focuses on finding the code aspects by means of an aspect-mining tool called FINT. The third step is to find the traceability links using both the early aspects and the code aspects findings. The EA-Tracer showed a high accuracy level of finding aspects through its two-fold method thus helping in aspect removal and resulting in lesser defects [11].

Figueiredo et al. [12] looked at metrics such as the Concern Diffusion over Components (CDC), Concern Diffusion over Operations (CDO), Concern Diffusion over Lines of Code (CDLOC), Dedication (DEDIC), Feature Crosscutting Degree (FCD), Lines of Concern Code (LOCC), Number of Concern Attributes (NOCA), SIZE, TOUCH, and FOCUS. They tested these metrics against different Business, Concurrency, Distribution, Exception Handling (EH), Persistence, and View concerns, to verify the effectiveness of the metrics. Their studies revealed more than 80% accuracy[12].

Rajan et al. [13] presented a language named Ptolemy to increase the separation of concerns with keeping encapsulation intact. Ptolemy has three advantages:

1. Enable refactoring and modularization of crosscutting concerns without affecting encapsulation [13].
2. Define an excellent interface between OO code and crosscutting concerns [13].
3. Allow OO code and crosscutting concerns to maintain separate type checking, compilation and modular reasoning [13].

Sun[14] presented an Aspect-Oriented modeling tool that deals with concern at the model level. The approach was applied successfully before writing any code [14].

Bernardi and Di Luca [15] proposed a tool to identify crosscutting Concerns in OO programs called ConAn. ConAn scans the system to find crosscutting relationships among the identified concerns to increase the number of crosscutting found[15].

4. FUTURE WORK

The area of detecting OO defects and crosscutting concerns is still in the theoretical phase. In other words, none of the popular development languages such as C#, Java, and C++ has any built-in or even add-in capability of detecting defects. A promising future work should concentrate on research directed to implement fully functional tools that can help in identifying the defects in early stages of development. These tools should become part of the debugging tools that remove compiling errors in programs.

5. CONCLUSION

Defect and crosscutting detection tools and techniques are very important in decreasing the number of flawed software and therefore increasing software quality. It is still very important to master the development tools and understand fully the different programming paradigms that developers and programmers are using. Defect prevention dictates that using the proper concept and tool to solve a problem leads to better results. Developers should learn when to apply the different programming paradigms in the context of the problem they are trying to resolve.
6. REFERENCES


