A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws

Mazeiar Salehie, Shimin Li, Ladan Tahvildari
Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1
{msalehie,s7li,ltahvild}@uwaterloo.ca

Abstract

One of the important activities in re-engineering process is detecting design flaws. Such design flaws prevent an efficient maintenance, and further development of a system. This research proposes a novel metric-based heuristic framework to detect and locate object-oriented design flaws from the source code. It is accomplished by evaluating design quality of an object-oriented system through quantifying deviations from good design heuristics and principles. While design flaws can occur at any level, the proposed approach assesses the design quality of internal and external structure of a system at the class level which is the most fundamental level of a system. In a nutshell, design flaws are detected and located systematically in two phases using a generic OO design knowledge-base. In the first phase, hotspots are detected by primitive classifiers via measuring metrics indicating a design feature (e.g. complexity). In the second phase, individual design flaws will be detected by composite classifiers using a proper set of metrics. We have chosen JBoss Application Server as the case study, due to its pure OO large size structure, and its success as an open source J2EE platform among developers.

1. Introduction

Evolution is an intrinsic property of software systems. As the software is enhanced, modified and adopted to new requirements, the code becomes more and more complex and drifts away from its original design. One of the first steps of maintaining an OO system is detecting design flaws, which may cause problems for future evolution/maintenance. Program comprehension plays a vital role through understanding the design of a software to detect drifts and flaws that may lead to such problems.

A design flaw, or according to Fowler [10] *bad smells* of design, mainly is a violation from one or more design principles. Riel [16] named these design principles *heuristics* and described that such rules should be thought of as a series of warning bells that will ring when violated. Tahvildari *et al.* [21] proposed a classification for design flaws related to the internal structure of a class, interactions among classes, and the application semantics. These three categories represent three non-orthogonal levels of abstraction, and thus may rely on different detection and correction techniques.

The main objective of this research is to define a framework to quantify design flaws at the class level relating to different features (*i.e.* complexity), and locate and isolate these flaws in a systematic manner for maintenance activities. This work could pave the way for providing a flaw diagnosis system like a medical diagnosis expert system in which by using symptoms, lab results and a rule set, specific diseases could be diagnosed. Such a medical system not only could be used to find causes of a pain, but also could test biometrics and find a hidden flaw or a potential future problem (i.e. a possible heart attack 5 years later due to a high level of cholesterol). In this paper, we deal with the second scenario when we start the work on a software system without a priori knowledge of any flaws. Such a novel framework can help two categories of maintenance, namely *preventive* which is performed to prevent problems before they occur, and *perfective* which is performed to modify a software product after delivery to improve maintainability.

The remainder of this paper is organized as follows. Section 2 describes the problem while Section 3 describes details of our proposed approach to detect design flaws in an object-oriented system. Section 4 describes the structure of proposed generic OO design knowledge-base. Section 5 describes the structure of primitive and composite classifiers when the process of detecting design flaws is applied. Section 6 presents and discusses the experimental results obtained by applying the proposed process on the target case study, JBoss Application Server 4.0. Section 7 reviews related works in the area of detection design flaws. Finally, Section 8 summarizes the contributions of this work and outlines directions for further research.

2. Problem Statement

By introducing mechanisms like inheritance and encapsulation of data, object-oriented design aims at quality factors such as maintainability and reusability. Applying these concepts do not always produce high quality software [16]. Although a large number of metrics have been proposed by researcher to measure OO quality factors, they have two major problems: i) there is a large gap between design principles and design metrics, and ii) there is no clue how the correction strategy can be applied to improve the quality after detecting flaws using metrics.

There is a gap in quality models (*e.g.*, Factor-Criteria-Metric (FCM) [4]) between quality factors and criteria from one side and metrics from the other side. In fact, it is a gap between subjective quality goals and objective quantity measurements. On the other hand, there are hidden semantics behind mapping from quality criteria to metrics, so it is not only hard to understand the relation between metrics and design rules, but also it is difficult to recognize what are the real causes of the symptoms. There are some works in literature which tried to propose quality models of OO software systems, but in spite of mitigating the mentioned gap, they couldn't either cover all the factors or clarify links between design rules, metrics and quality factors [1, 13, 18].

The aforementioned gap has a significant impact on detecting design flaws. This is due to the core concepts of design flaws: measuring violations/deviations from design principles using metrics, and simultaneously quantify quality factors such as maintainability. In this paper, we propose a generic OO design knowledge-base encompassing major design heuristics/principles and metrics to detect design flaws. In fact, design features [23] (e.g. coupling), have the key role in building the proposed OO design knowledge-base and filling the gap between subjective and objective concepts. Although the proposed approach is extensible and configurable, the design flaws, the OO metrics, and the heuristic rules presenting in this paper only addresses one quality attribute namely, *maintainability*.

3. A Proposed Approach

To come up with a systematic solution to address the problem, we return to the medical diagnosis process and see how a physician monitors the degree of healthiness of a patient. A physician has a knowledge of medicine encompasses normal range for different biometrics such as blood pressure, the number of heart beating in a minute, amount of glucose, cholesterol and several other important factors. The second significant information in the knowledge-base of a physician are symptoms and the quantity of biometrics for various diseases or possible diseases in the future (for instance high amount of cholesterol as a symptom for a potential heart attack in the future). The diagnosis process for a physician is testing major biometrics at first, and in the case they are not in normal range she may need to measure more biometrics to compare with her knowledge about diseases.

In our approach, we want to emulate the same process for a software system by regarding the differences between a human and a software system as well as the lack of the perfect knowledge about normal/expected values (thresholds) for most of the software metrics. We assume that there is no a priori knowledge about the design of a system and all design facts will be extracted and measured from the source code. As depicted in Figure 1, our approach encompasses the following components:

- A *generic OO design knowledge-base* containing design heuristics, metrics, flaws and their relationships. The structure of the generic design knowledge-base is flexible enough to customized for different set of design flaws.

- A *hot spot indicator* pointing out the most probable defective entities, namely "hot spots" using *primitive classifiers*.

- A *design flaw detector* locating possible design flaws in the predetermined hot spots using *composite classifiers*.

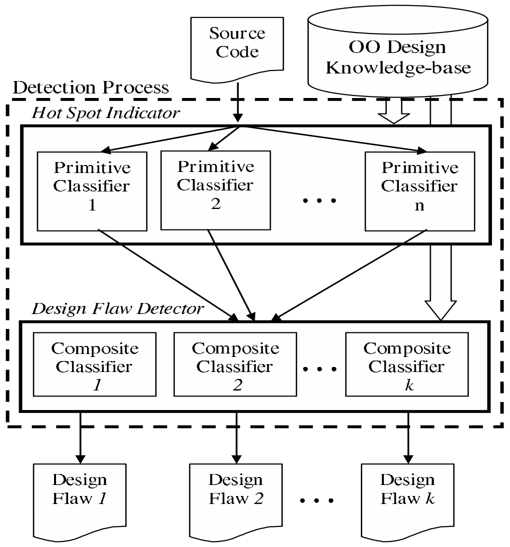
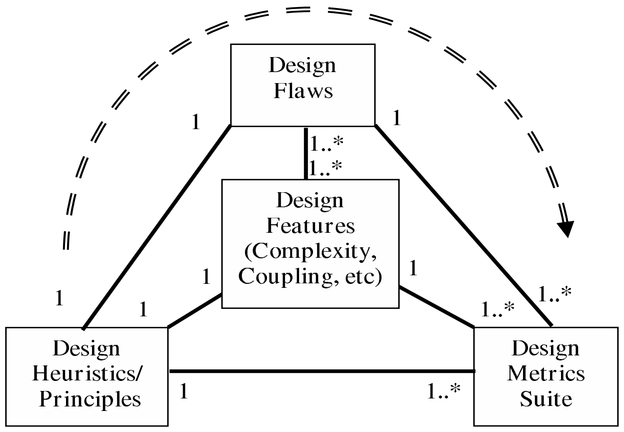


Figure 1. The Architecture of the Framework to Detect OO Design Flaws.

Based on the target set of design flaws, the design knowledge-base should be built and then classifiers at both levels should be defined for performing the automated detection mechanism. While Section 4 elaborates further on the characteristics of such generic structure of the knowledge-base, Section 5 discusses the detection process.

4. A Generic OO Design Knowledge-Base

As depicted in Figure 2, the proposed generic OO design knowledge-base encompasses four major entities namely *design heuristics/principles*, *design flaws*, a *design metric suite* and *design features*. The OO design knowledge-base entities are either subjective (*e.g.*, design flaws) or objective (*e.g.*, metrics). Design features have been used as a glue to relate subjective to objective entities. The detection process needs such a design knowledge-base because: i) we need to fill the gap between qualitative and quantitative measures, and ii) metric-based measurements are often hard to interpret and normally do not show the root cause of a flaw.



Design Flaws
Design Features (Complexity, Coupling, etc)
Design Heuristics/ Principles
Design Metrics Suite
1
1
1..*
1..*
1
1
1
1
1..*
1..*
1
1..*

Figure 2. A Generic OO Design Knowledge-Base Model

The model of the generic OO design knowledge-base contains predefined but customizable catalogues for heuristics, flaws, features, and metrics. For any system, first of all, the OO design knowledge-base must be configured. The order of configuring is shown by a dashed arc in Figure 2. The configuration begins with selecting the design heuristics/priciples as the root of design flaws. In fact, design flaws are violations of these heuristics/principles. Then the target design flaws are defined in terms of the features in the design knowledge-base. Finally, the features will be related to selected metrics in the metric catalogue. The details are elaborated further as follows :

STEP 1: Defining Design Heuristics/Principles
This entity consists of principles and heuristical rules for good design in object-oriented systems. Object-oriented design principles are mostly extensions of general design principles in software systems (*e.g.*, abstraction, modularity, information hiding). Samples of principles for good design in software systems are: *high coupling*, *low cohesion* and *complexity*. Design heuristics [16] are stated as the rules of thumb or guidelines for good design. These rules are based on design principles and their ultimate goal is to improve quality factors of the system and avoid occurrence of design flaws. These rules recommend designers and developers to "do" or "do not" specific actions or designs. A sample of such heuristics is "minimize the number of messages in a class". Although, they may not be directly defined as rules based on design features, they can be related to one or several features. In fact, principles/heuristics recommend or oppose high or low degrees of one or combination of several design features. This research extensively uses design heuristics presented in [16] to build the proposed entity.

STEP 2: Mapping Design Flaws to Heuristics
In the recent years, we found various forms of descriptions for bad or flawed design in the literature such as *bad-smells* [10]. Riel [16] presents a set of heuristic design guidelines, and discusses some of the flawed structures that result if these guidelines are violated. In the same manner, Martin [14] discusses the main design principles of object-orientation and shows that their violation leads to a *rotting design*. The first prerequisite for detecting such flaws is to have classification rules for each flaw. Each flaw should be defined by such rules based on the different design features of the source code. On the other hand, for applying these classification rules, quantitative measures are required which can be defined by design metrics. As depicted in Figure 1, two levels of classification are considered in our proposed framework which require two set of classification rules. Fowler *et al.* [10] give descriptive definitions for design flaws such as *Shotgun Surgery* and *Data Class*. The definition given by Riel [16] are better for defining classification rules because he not only defines flaws based on design heuristics but also categorizes design heuristics which can easily map to design features. Table 1 shows a subset of such design heuristics/priciples and their mappings to design flaws at the different levels of granularity in OO systems.

STEP 3: Mapping Design Flaws to Design Features
Design features such as complexity are the core concept of the generic OO design knowledge-base that is why all three major entities in the proposed OO design knowledge-base explicitly or implicitly relate to these features as shown in Figure 2. For instance design heuristics/principles deal with managing complexity, while some design flaws like God Class concern too complex classes. On the other hand, there is a category of metrics to measure the complexity at different levels of a software system. In this way, the classifying rule can identify the hot spots which ultimately wave to design flaws using these features. Depending on the selected design flaw and the level of abstraction some of these features may be decomposed to sub-features (e.g. decomposing complexity to inter- and intra-class complexity). Using these categories, we can define features as the

Abstraction Level	Heuristics/Principles	Design Flaw
Class	i) Distribute system intelligence horizontally as uniformly as possible, ii) Beware of classes that have many accessor methods defined in their public interface, and iii) Do not create God Classes/Objects in your system [16].	**God Class**: One class is used more extensively than others [16].
Class	Minimize the Number of Messages in the protocol of a class [16].	**Shotgun Surgery**: every time you make a change in a class; you have to make many tiny changes in lots of different classes [10].
Class	Factor the commonality of data, behavior, and/or interface as high as possible in the inheritance hierarchy [16].	**Refused Bequest**: Subclasses do not want or need everything they inherit [10].
Method	Keep related data and behavior in one place [16]	**Feature Envy**: Often a method that seems more interested in a class other than the one it's actually in [10].
Method	Manageable Complexity [16]	**God Method**: One method is used more extensively than others [16].

Table 1. A Subset of Mapping Design Heuristics/Principles to Flaws.

joint points of three other components of the proposed OO design knowledge-base as depicted in Figure 2. Applying such an approach, each design flaw is defined as a classification rule using design features based on violation of design heuristics/principles. Table 2 shows the mapping between a subset of design flaws and features. By measuring the specified features for each flaw, it is possible to detect hot spots and a set of flaws.

Design Features/ Design Flaws	God Class	Shotgun Surgery	Refused Bequest	Feature Envy	God Method
Complexity	√	√			√
Coupling	√	√		√	
Cohesion	√		√	√	
Inheritance			√		

Table 2. Mapping Design Flaws to Features

Generally, the number of design features are limited. So defining classification rules in terms of design features are easier than defining them directly by metrics. By this mean, flaws can also be directly related to design heuristics. Classification rules can be defined by degrees of each features. An appropriate way is to use fuzzy terms which are closer to human linguistic terms and so are more comprehensible for defining the OO design knowledge-base. Moreover, Each feature may be measured by different metrics for two different design flaws or in primitive and composite classifiers of a specific flaw.

STEP 4: Defining a Catalogue of Design Metrics
Design metrics quantify design features of specified flaws in a OO design knowledge-base. These metrics are selected based on the definitions and classification rules of each flaw. Because design features are the core of the OO design knowledge-base model, there is a flexibility of using different metrics for different levels of abstraction in a system (class or method level) or different systems. To select a metric, the important question is that whether the metric is a valid quality indicator for the specified design feature (e.g. complexity). Chidamber-Kemerer [7] proposed a metric suite for OO systems in four categories (coupling, cohesion, inheritance and size) and after that several works were published on analyzing validity of these metrics and the modified metrics as quality indicators. Most of the researchers used logistic regression and principal component analysis for this purpose. Basili *et al.* [2] showed 5 out of the 6 Chidamber-Kemerer metrics could be good quality indicators. Ping Yu *et al.* [20] analyzed 10 metrics in 5 categories, the former categories and reuse, and showed 7 metrics could be significant in the system quality. Fioravanti *et al.* [9] also selected 12 metrics in 4 categories (same as Chidamber-Kemerer) out of 200 proposed metrics in literature. Briand *et al.* [5] built a fault prediction model including more than 6 metrics from 3 categories (coupling, cohesion and inheritance). Industrial projects and commercial tools also have categorized metrics in different manners, for instance Datrix has four categories of routine, class, file, and Halstead metrics. Borland Together uses 12 categories for object-oriented metrics including complexity, coupling, inheritance and cohesion.

Design Feature	Metric	Classifier
Coupling	CBO (Coupling Between Objects)	Primitive
Cohesion	TCC (Tight Class Cohesion)	Primitive
Intra-class Complexity	WMPC (Weighted Method Count)	Primitive
Inter-class Complexity	RFC (Response For a Class)	Primitive
Coupling	AOFD (Access Of Foreign Data)	Composite
Coupling	CM (Changing Methods)	Composite
Coupling	ChC (Changing Classes)	Composite

Table 3. A Subset of Metrics Used in OO Design Knowledge-Base

Table 3 lists a sample metric suite for the proposed OO design knowledge-base. The third column of Table 3 will be elaborated further in Section 5. Validity of metrics, relationship to specified design features, and availability of a method/tool for measurement are our criterion for selecting the metric suite in the proposed framework. Some metrics are selected as a subset of metrics presented in [2] based on the existing hypotheses in OO systems (like normal range in biometrics). These hypotheses maps to the design heuristics that of the interest of this research work. The rational behind the selection of these metrics are discussed as follows:

- *Coupling Metric*: We selected CBO (Coupling Between Objects) [7] as the primitive metric for coupling. CBO provides the number of classes to which a given class is coupled by using their member functions and/or instance variables. As shown in [2, 20], it is a significant quality indicator for OO systems. AOFD (or ATFD) also represents the number of external classes from which a given class accesses attributes, directly or via accessors methods. CM (Changing Methods) is defined as the number of distinct methods in the system that could be affected by changes in the measured class. The methods affected are all those that access an attribute and/or call a method and/or redefine a method of the given class. ChC (Changing Classes) measures the number of client-classes which must be changed as the result of a change of the server-class.
- *Cohesion Metric*: LCOM (Lack of Cohesion in Methods) [3] is not a significant cohesion indicator as discussed in [2, 11, 20]. We have chosen TCC (Tight Class Cohesion) as indicated in [9]. TCC is defined as the relative number of directly connected methods. Two methods are directly connected if they access a common instance variable of the class. TCC refers the relative number of directly connected methods in a given class.
- *Complexity Metric*: We use two categories of complexity [23]: i) *Intra-class Complexity* (inherent complexity) which is close to cyclomatic complexity, and ii) *Inter-Class Complexity* (interaction complexity) which in fact is the intersection of complexity and coupling. We have chosen WMPC (Weighted Method Count) for the first category and RFC (Response For a Class) for the second one. WMPC is the sum of the complexity of all methods for a class, where each method is weighted by its cyclomatic complexity. The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class. RFC is the number of methods that can potentially be executed in response to a message received by an object of that class. The reason for selecting RFC is based on the nature of its definition and high correlation with WMPC (Weighted Method Count) and CBO as two major metrics for complexity and coupling. Systa *et al.* [20] show this correlation, and our results for our case study (JBoss) verify this assertion.

5. A Detection Process

As depicted in Figure 1, the proposed framework to detect design flaws contains a two-level classification process. The first level aims at indicating hot spots based on a set of primitive classifiers. The second level aims at detecting design flaws by classifying the hot spots through more precise analysis and filtering. Because the detection rules for design flaws contain a level of uncertainty and possibility, finding these flaws in one-pass classification using certain metric-based rules cannot be considered as a perfect approach. Two passes of classification with two different rule set can help us to detect potential and highly potential problematic entities in a system through a systematic way. At the first level, the emphasize in on the metrics with strong relationship with features, while at the second level the metrics which address overlap of the features are also taken into account.

5.1. Hot Spot Indicator

As shown in the architecture of the proposed framework, the primitive classifiers in the hot spot indicator determine the entities (classes, methods, etc) which may be problematic. Detecting these potential flaws are accomplished by primitive rules defined in the OO design knowledge-base for the specified design flaws. One example of such primitive rules is (in a pseudo format):

```
if (class X has high Inter-Class Complexity) and
   (class X has high Coupling)
then (class X probably has Shotgun Surgery)
```

Here we use the fuzzy terms *high* and *low* to classify classes to each of the aforementioned categories. We believe that fuzzy terms are appropriate for design features and metrics in classifiers because: i) they can be configurable and flexible for different systems, and ii) they reflect the uncertainty about the precise value of the thresholds and possibility factors in detecting flaws. The nature of hot spots, as the potential spots of flaws, also supports the second assumption. We need threshold values to quantify these high and low terms. These thresholds can be relative (e.g. top 10 values) or absolute (e.g. more than 5). It is hard to set a strict and crisp value for all OO systems, although some researchers proposed an "industrial average" for several metrics, but still there is no consensus on proper values for thresholds. Relative thresholds seem to work better, but they cannot be used individually because they always select for instance top 20 values and we have always a constant number of hot spots.

5.2. Design Flaw Detector

After determining the hot spots, the goal is to detect the final set of problematic entities. Design flaw detector is also based on metric-based classifiers (called *composite classifiers*) which use combinations of metrics mostly different from primitive metrics to measure more details about the suspicious entities. This is similar to the process by which a physician finds potential problems in the first phase by simple biometrics such as sound of the heart, and then checks the detail by more accurate metrics like electro-cardiogram. For each design flaw, there is a composite classifier to analyze the details of hot spots and figure out whether they have a flaw. The metrics selected for primitive classifiers are the minimal set of metrics strongly related to their assigned design features. But to improve the accuracy of the measurement in composite classifiers, we need to measure each design feature probably by several metrics. On the other hand, design features are not completely independent; so to cover the overlap among them we also need additional metrics. In fact, the idea of using two-level classifiers are suitable for large-scale object-oriented systems. In the primitive level by using less complex rules suspicious entities are filtered. Then a combination of more complex metric-based rules can filter the first set. An example rule for composite classifiers can be considered as follows :

if *(class X has high CM)* ***and*** *(class X has high ChC)*
then *(class X has Shotgun Surgery Flaw)*

6. Case Study: JBoss Application Server

In this section, we apply the proposed heuristic-metric framework on an industrial large-scale case study, namely JBoss Application Server 4.0 [1]. JBoss is an open source, standards-compliant J2EE application server implemented in Java and distributed for free under the LGPL license. It is the most downloaded application server in the world based on the J2EE specification. As shown in Figure 3, JBoss Application Server is a part of what we call middle-tier, between the end-user tier and the primary services. JBoss Application Server 4.0 is composed of 28 subsystems hierarchically organized. The JBoss system has 952 KLOC with 6058 Java files and 4892 classes. It has three layers - Microkernel, Service and Aspect Layer - in its architecture as illustrated in Figure 3 (interested readers can refer to [19] for more information regarding its extracted architecture). We used Borland Together for Eclipse [2] to measure metrics and Microsoft Excel for statistical and logical operations.

6.1. Configuring the Design Knowledge-Base

For this case study, we have selected two flaws at the class level namely : *God Class* and *Shotgun Surgery*. The first step to exploit the framework is configuring the OO design knowledge-base for these selected design flaws. These flaws are related to architectural and structural categories (besides of intermediate categories) [21]. We use a subset

[1]http://www.jboss.org
[2]http://www.borland.com

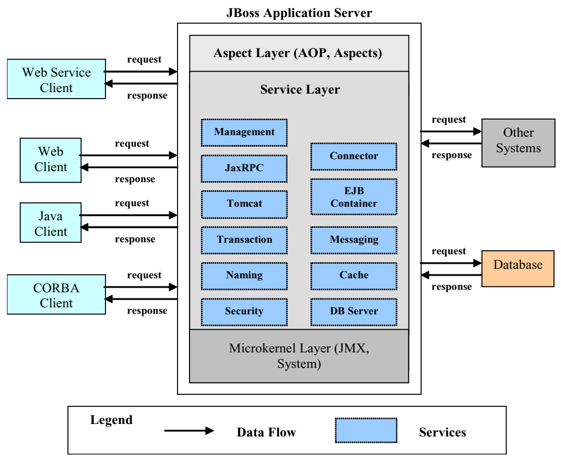


Figure 3. JBoss Application Server Layers.

of the sample catalogues defined in the knowledge-base (discussed in Section 4), for taking into account a set of metrics related to four features namely intra- and inter complexity, cohesion and coupling. The rationals behind of such selections in terms of flaws and metrics are elaborated further as follows :

God Class: A class does most of the functionalities of a system in the comparison with other classes in the system contributes to the God Class smell. In a nutshell, god Class refers to those classes which tend to centralize the intelligence of the system. An instance of a God Class performs most of the operations, delegates only minor details to a set of trivial classes, and uses the data from other classes. This design flaw is partially analogous to Fowler's Large Class smell [10]. Hence, God Classes deviate from the heuristic of manageable complexity, low coupling as well as tend to be also non-cohesive. To detect a God Class, we look for classes : i) use a lot of data from the other classes, either being highly complex or having a large state, and ii) have low cohesion between methods. as shown in Table 4 and Table 3, the defined God Class detection rule should measure four design features of a given class: intra- and inter-complexity, cohesion, and coupling. For detecting this flaw, CBO, TCC and WMPC are used in the primitive classifier and AOFD, WMPC and TCC metrics are used in the composite classifier.

Shotgun Surgery: A class that is coupled to a large number of other classes, and would produce a large number of changes throughout the system in the event of an internal change, contributes to the Shotgun Surgery smell. Shotgun Surgery means that a change in a given class implies many changes to a lot of different classes. By the definition, a class which presents the Shotgun Surgery flaw tends to be coupled to a large number of other classes. In a nutshell, Shotgun Surgery deviates from the heuristic of minimization the number of messages in the protocol of the class. As shown in Table 4 and Table 3, for detecting this flaw, CBO and RFC are used for the primitive classifier and ChC and CM metrics are used for the composite classifier.

6.2. Building Classifiers

According to the definition of the selected design flaws, we need to classify four kind of classes: i) *intra-class complex* to find classes which are individually complex not regarding their communications with the other classes, ii) *inter-class complex* to find classes which have high level of responsibility in communication with other classes, iii) *highly-coupled* to find classes which are highly coupled to others, and iv) *low-cohesive* to find classes that are not cohesive enough.

First, the primitive classifiers labels classes as intra-class complex, inter-class complex, high-coupled, low-cohesive, or a combination of them to find hot spots. To build the primitive classifiers, we relate these design flaws to proper design features, and then we propose metric-based rules to detect classes with these flaws using the predefined mappings.

Design Features/Flaws	**God Class**	**Shotgun Surgery**
Intra-class Complexity	High	
Inter-class Complexity		High
Coupling	High	High
Cohesion	Low	

Table 4. Mapping Design Flaws to Features Using Heuristics

Table 4 shows the mapping between selected design flaws and features with assigning linguistic terms. In fact, "High" shows positive correlation and "Low" shows negative correlation between design features and metrics. Based on the mapping, the metric-based rules are used for primitive classifiers as follows : i) High-coupled are classes with high level of CBO, ii) Intra-class complex are classes with high level of WMPC, iii) Inter-class complex are classes with high value of RFC, and iv) Low cohesive are classes with low level of TCC.

Similar to primitive classifiers, composite classifiers are built by mapping between flaws to metrics through features. Detection rule for God Classes according to Table 2 and 3 can be formulated as follows:

if *(class X has high [AOFD* **&** *WMPC* **&** *TCC])*
then *(class X has God Class Flaw)*

The classification rule for Shotgun Surgery can be also formulated as follows :

if *(class X has high [CM* **&** *ChC])*
then *(class X has Shotgun Surgery Flaw)*

6.3. Obtained Results

This section discusses our findings about design quality of JBoss Application Server 4.0 using the proposed framework. Due to the large scale of our case study , and the space constraints in this paper, we only present a subset of our results. Interested readers could refer to [19] for more details. We have applied our two-level classification approach on 4297 classes of JBoss, after eliminating some third-party subsystems such as Tomcat from our analysis.

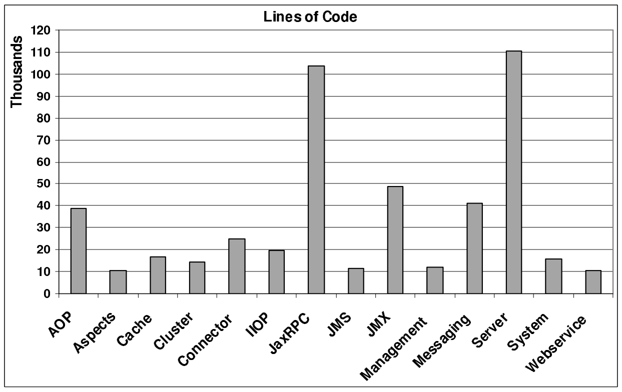


Figure 4. JBoss Subsystems LOC

First, we were interested to analyze the relationships between the size and the number of detected design flaws in each subsystem of the case study. Figure 4 illustrates sizes of the subsystems in JBoss. A depicted in Figure 5, the five top large size subsystems have more flaws and when we analyzed the classes inside of each subsystems there was a correlation between size and flaw in a class. For instance in *Server* subsystem, there were 17 problematic classes (either having God Class or Shotgun Surgery flaw) out of top 20% large classes (162 out of total 812 classes). All of the 6 God Classes in the *Server* subsystem were in the top 20%, This was not a surprise for us due to the relation between complexity and the module size. But the class or subsystem size does not show a precise predictive clue about the possibility of flaw or fault-proneness in the same module. Such a conclusion emphasizes that we cannot specify a threshold or the optimal class size in an object-oriented system.

Another interesting point for us was to study the relationships between design flaws. Figure 5 shows the distribution of design flaws in JBoss subsystems (except of packages without any design flaw). As illustrated in Figure 5, we have only 3 coincidences of these two flaws in a package in all 14 packages (21%) and only one class has been found in the JBoss that has simultaneously both flaws (JDBCUtil class in the *Server* subsystem).

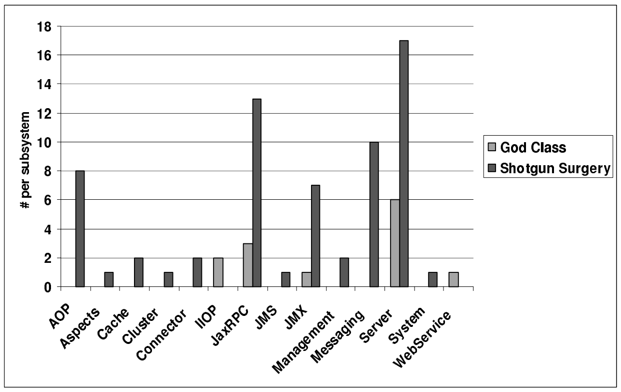


Figure 5. Detected Design Flaws in JBoss

We posted the results to JBoss developer forum to know their opinion about these findings. In fact, they had not analyzed JBoss from this point of view by these definitions for design flaws, but they asserted that in the flaw list there are many instances of problematic classes, regarding their design, which they modified them during maintenance/evolution process. One interesting case was JMSContainerInvoker class, which one of chief scientists of JBoss believed it could contain a design flaw, probably shotgun surgery, but it was not in our list. The logs shows that this class has been detected as a hotspot by the primitive classifiers mainly due to its high coupling with other classes. But, it has been rejected as a design flaw by composite classifiers, because for shotgun surgery the important factor is that how much other classes rely on this class. So, having a high-level of coupling does not necessarily mean a class has the shotgun surgery flaw.

To study the correlation of metrics in our metric suite, we calculate correlation of these metrics in JBoss. Table 5 shows the calculated correlations. We assumed values above 0.50 as correlated metrics (shown as bold numbers in Table 5). There is a positive correlation between RFC and CBO which in fact is between inter-class complexity and coupling. On the other hand, RFC and WMC are highly related and this is because they both are indicating complexity. These values shows that RFC and consequently inter-class complexity feature is in the middle of coupling and complexity, but it shows facts that cannot be shown by either selected coupling or intra-class complexity. AOFD and CBO also show correlation and this fact justifies selection of AOFD as the second phase indicator of coupling in the God Class classification rule.

For evaluating the proposed process and comparing the results with the related works, we use two criteria of precision and recall for the hot spot and final sets of each detected design flaws. Precision is the proportion of correct detections to all of the detected cases, and recall is the proportion of correct detected cases to all of the cases in the system. If *DF*, *HS* and *FF* are respectively the set of Design Flaws, Hot Spots and the Final Flaws of detected design flaws in the systems, we define the precision and recall at the first and second phases of the detection process based on the cardinality of these sets as follows :

$$Hot\ Spot\ Precision\ =\ |HS\ \cap\ DF|/|HS|$$
$$Hot\ Spot\ Recall\ =\ |HS\ \cap\ DF|/|DF|$$
$$Final\ Set\ Precision\ =\ |FF\ \cap\ DF|/|FF|$$
$$Final\ Set\ Recall\ =\ |FF\ \cap\ DF|/|DF|$$

Using above formulas, we can compare our results with the strategy proposed in [12, 13] and show that our approach is much more flexible and configurable for large-scale object-oriented systems. Hot spots for the God Class flaw is the intersection of three sets namely: high-coupled, low-cohesive and intra-class-complex. The hot spot set consists of 533 classes out of 4296 total classes. Table 6 shows the statistics of applying the approach for detecting God Classes. The precision and recall have been calculated for hot spots and the final set of God Classes.

Stats / Design Flaw	God Class	Shotgun Surgery
Hotspots	533	1510
Hot Spot Set Precision	2%	3%
Hot Spot Set Recall	100%	74%
Final Set Precision	100%	94%
Final Set Recall	100%	74%

Table 6. Selected Results on Design Flaws

Hot spots for the Shotgun Surgery design flaw is the intersection of two sets namely: high-coupled, inter-class-complex. As shown in Table 6, the obtained results show 1510 hot spots out of 4296 total number of classes. We found 197 cases with absolute thresholds rules (CM $>$ 10 and ChC $>$ 5) which top 25 percent gave us 51 classes with Shotgun Surgery flaw. Our method have not found 17 classes which were directly found with the strategy proposed in [12] while 3 cases (5%) is additional. The hot spot indicators mostly focus on increasing recall to capture most of the potential problematic classes for the next phase. By applying the second level classifiers for each design flaw, precision increases which decreases the number of additional cases in comparison with [12].

The results for the selected case study, JBoss Application Server 4.0, show that our proposed framework can systematically and efficiently detect design flaws of an object-oriented system. It is efficient because it does not check all the classes in a large system by applying complex rules, and giving hard-to-interpret results. It finds hot spots by quick classifications in the first phase and then focuses on the hot

	CBO	TCC	WMC	RFC	AOFD	CM	ChC
CBO	1	0.0497	**0.6162**	**0.7646**	**0.6357**	0.1767	0.2410
TCC	0.0497	1	0.1064	0.0872	0.0806	0.0100	-0.0001
WMPC	0.6162	0.1064	1	**0.9160**	0.4852	0.2369	0.2587
RFC	0.7646	0.0872	0.9160	1	**0.5404**	0.2172	0.2456
AOFD	0.6357	0.0806	0.4852	0.5404	1	0.1019	0.1558
CM	0.1767	0.0100	0.2369	0.2172	0.1019	1	**0.8259**
ChC	0.2410	-0.0001	0.2587	0.2456	0.1558	0.8259	1

Table 5. Correlation among Selected Metrics in JBoss Application Server 4.0

spot set to find the final set of design flaws by more precise and complex rules. One of the benefits of this approach for a future flaw diagnosis expert system is the capability of giving clear and easy-to-understand reasons for questions like : "Why the diagnosis system has chosen a class as a God Class?"

Two important advantages of our approach are the extensibility and configurability of the proposed framework. Our proposed approach can be extended : i) by adding more quality factors, their related heuristics/principles, design features, design flaws and metrics to the OO design knowledge-base, and ii) by building the appropriate classifiers for the hot spot indicator and the design flaw detector using a set of primitive and composite classifiers.

7. Related Works

During the past years, various approaches have been developed to address the problem of detecting and correcting of design flaws in an OO software system. Marinescu [12] defined a list of metric-based detection strategies for capturing around ten flaws of OO design at method, class and subsystem levels as well as patterns. Since the interpretation of individual measurements of metrics is too fine-grained, Marinescu introduced a filtering and composition mechanism to find design fragments that are affected by a particular design flaw. However, how to choose proper threshold values for metrics and propose design alternatives to correct the detected flaws are not addressed in his research.

Mihancea *et al.* [15] presented an approach for establishing proper threshold values for metrics-based design flaw detection mechanism. This approach, called *tuning machine*, is based on inferring the threshold values based on a set of reference examples, manually classified in flawed respectively good design entities. They tried to optimize God-class detection strategy using a genetic algorithm. Their work toward finding the optimum threshold values led to multiple threshold sets which can be applied in parallel and the majority will be determined the final set of God Classes. Another interesting approach to increase the accuracy is to enhance the detection process by combining detection strategies applied on a single version with additional information extracted from multiple versions of the system [17].

Emden *et al.* [8] worked on software inspection to repair bugs, called code smells, early in the software development cycle to decrease the total cost. They discussed that such code smells are subjective, domain-dependent and not precise in terms of underlying parameters. They developed jCosmo which is configurable to add or remove or change definition of smells. Brown *et al.* [6] introduced the concept of anti-pattern in contrast with design patterns as the frequently observed bad solutions at higher levels of design, the class level or higher.

Recently, some literature tried to bridge the gap between detection and correction of design flaws. Tahvildari *et al.* [21] investigated the use of OO metrics for detecting potential design flaws and presented a correction mechanism. The correction mechanism is based on analyzing the impact of various meta-pattern transformations on these OO metrics. Trifu [22] introduced correction strategies based on the existing flaw detection and transformation techniques. This approach serves as reference descriptions that enable a human-assisted tool to plan and perform all necessary steps for the removal of detected flaws. Consequently, it is a methodology that can be fully supported.

However to the best of our knowledge, the efficiency and accuracy of all current design flaw detection and correction strategies for large-size industrial OO systems (above 800 KLOC) have not been assessed where our proposed approach addresses such design flaws detection. The proposed two-phase detection mechanism is flexible to modify the classification rules, and easily expandable to detect more design flaws.

8. Conclusions and Future Work

This paper proposes a framework for detecting object-oriented design flaws. An OO design knowledge-base of related design principles, metrics and design flaws concepts needs to be developed. The core part of this knowledge-base is design features which binds all the other three concepts. For this paper, we have considered only a part of this OO design knowledge-base at the class level, and proposed the heuristics for both hot spot indictor and design flaw detector which are based on the underlying classifiers. Classifiers use category-based rule sets, which in turn are metric-based. So, the process actually is a two-level classification process to detect the final flaws in the system. The proposed framework begins the process by defining design flaws in terms of well-established principles, features and metrics instead of proposing a magic formula. This configurable knowledge-base helps experts to define classifiers and to interpret results in a systematic manner. The two-phase classification also eases the detection process in addition of finding hot spots which may not have design flaw at the current state but potential to have one in the future.

There are three noteworthy points about our proposed framework that can be considered as future work : i) it will be more appropriate to have different weights in classifying various design features instead of equal weights that have been used in the approach, ii) it will be useful to include a degree of possibility or a kind of certainty factor for the heuristics and the detected design flaws as we cannot specify strict threshold values for ”high” or ”low” terms in classifiers rules, and iii) a ranking mechanism for design flaws could be useful for developers and maintainers to deal with high priority and critical flaws. An aggregation formulation for different flaws in different levels of abstraction could be appropriate for this purpose. Another possible way to improve the quality of results is adding a mechanism of evaluation for the final sets in terms of accordance with the design documents, but there is no direct and objective way to do this task. One of the other strategies that could be useful for some cases is to compare several successive versions. If so-called flaws repeated in more than two releases, then it could be a design decision not a flaw. Number of releases that should be analyzed to infer such a result may depend on the type of a flaw or the scale of the system.

References

[1] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17, 2002.

[2] V. Basili, L. Briand, and W. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22:751–761, October 1996.

[3] J. Bieman and B. Kang. Cohesion and reuse in an object-oriented system. In *Proc. ACM Symposium on Software Reusability*, pages 259–292, April 1995.

[4] B. Boehm, J. Brown, , and J. Kaspar. Characteristics of software quality. *TRW Series of Software Technology*, 1978.

[5] L. C. Briand, J. Wust, S. V. Ikonomovski, and H. Lounis. Investigating quality factors in object-oriented designs: an industrial case study. In *Proc. of the 21st int. conf. on Software engineering (ICSE)*, pages 345–354, 1999.

[6] W. J. Brown, R. C. Malveau, I. Hays W. McCormick, and T. J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, 1998.

[7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[8] E. V. Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proc. of the Ninth Working Conf. on Reverse Engineering (WCRE)*, pages 97–106, 2002.

[9] F. Fioravanti and P. Nesi. A study on fault-proneness detection of object-oriented systems. In *Proc. of the Fifth European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 121–130, 2001.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, 1999.

[11] H. Kabaili, R. K. Keller, and F. Lustman. Cohesion as changeability indicator in object-oriented systems. In *Proc. of the 5^{th} European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 39–46, 2001.

[12] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proc. of the IEEE 20^{th} Int. Conf. on Software Maintenance (ICSM)*, pages 350–359, Chicago, USA, September 2004.

[13] R. Marinescu and D. Ratiu. Quantifying the quality of object-oriented design: The factor-strategy model. In *Proc. of the 11th Working Conf. on Reverse Engineering (WCRE)*, pages 192–201, 2004.

[14] R. Martin. Acyclic visitor. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pages 93–104. Addison-Wesley, 1998.

[15] P. F. Mihancea and R. Marinescu. Towards the optimization of automatic detection of design flaws in object-oriented software systems. In *Proc. of the 9^{th} European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 92–101, 2005.

[16] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[17] D. Rtiu, S. Ducasse, G. T., and M. R. Using history information to improve design flaws detection. In *Proc. of the IEEE 8^{th} European Conf. on Software Maintenance and Re-engineering (CSMR)*, pages 223–232, Tampere, Finland, March 2004.

[18] H. A. Sahraoui, M. Boukadoum, and H. Lounis. Building quality estimation models with fuzzy threshold values. *LObjet*, 17(4):535–554, 2001.

[19] M. Salehie, S. Li, and L. Tahvildari. Architectural recovery of jboss application server. Technical Report UW-ECE-2005-02, University of Waterloo, January 2005.

[20] T. Systa, P. Yu, and H. Muller. Analyzing java software by combining metrics and program visualization. In *Proc. of the Conf. on Software Maintenance and Reengineering (CSMR)*, pages 199–208, 2000.

[21] L. Tahvildari and K. Kontogiannis. Improving design quality using meta-pattern transformations : A metric-based approach. *Journal of Software Maintenance and Evolution : Research and Practice*, 16(4–5):331–361, July–October 2004.

[22] A. Trifu, O. Seng, and T. Genssler. Automated design flaw correction in object-oriented systems. In *Proc. of the IEEE 8^{th} European Conf. on Software Maintenance and Re-engineering (CSMR)*, pages 174–183, Tampere, Finland, March 2004.

[23] H. van Vliet. *Software engineering: principles and practice*. John Wiley & Sons, 2000.