Analysis of Software Architecture Quality Metrics

Hongyu Chen

April 10, 2014
I, Hongyu Chen, hereby affirm in lieu of oath that the Master Thesis at hand is my own written work and that I have used no other sources and aids other than those indicated. All passages quoted from publications or paraphrased from these sources are properly cited and attributed.

The thesis was not submitted in the same or in a substantially similar version, not even partially, to another examination board and was not published elsewhere.

Aachen, April 10, 2014

(Hongyu Chen)
Abstract

Software architecture design is an important phase of software development. The quality of the initial software architecture has a big impact on the quality of the final software. Therefore, evaluating the quality of software architecture is necessary and an efficient way to improve the quality of a software. There have existed many metrics for measuring the quality of both software and software architecture in static view, but fewer metrics measuring that in behaviour view at runtime. Therefore, I proposed three kinds of metrics, Component metrics, Execution Hot Spot metrics and Violation metrics for evaluating the quality of software architecture in behaviour view. Component metrics containing Simple Component Cohesion Metric (SCCM), Hybrid Component Cohesion Metric (H.C.M) and Call-Weighted Hybrid Component Cohesion Metric (CW-H.C.M), measures the cohesion of a component design. Execution Hot Spot Metrics were made up by four metrics focusing on method hot spot and class hot spot. It was designed for optimizing the software architecture. Violation metrics were used to rank the results of communication rules checking, so that some suggestions can be put forward for solving the violations in a software.
## Contents

1. Introduction .................................................. 1

2. Background .................................................... 3
   2.1. Terminology .............................................. 3
   2.2. 4+1 view model .......................................... 4
   2.3. Static view of software architecture ...................... 5
   2.4. Behaviour view of software architecture ................... 7
   2.5. Behaviour view VS Static view .......................... 8
   2.6. Motivation ............................................... 12

3. Related work .................................................. 13
   3.1. Quality characteristics .................................. 13
   3.2. Static metrics ............................................ 15
   3.3. Static analysis tools ..................................... 19
   3.4. Dynamic metrics .......................................... 22
   3.5. Java profilers ............................................. 24

4. Concept ......................................................... 25
   4.1. Problems with the existing metrics ....................... 25
   4.2. Component metrics ....................................... 26
   4.3. Execution Hot Spot metrics .............................. 42
   4.4. Violation metrics ........................................ 46

5. Evaluation ..................................................... 57
   5.1. Data collection tools ..................................... 57
   5.2. CommunicationIntegrityChecker evaluation ............... 59
   5.3. JHotDraw 7 evaluation ................................... 68
   5.4. Summary .................................................. 72

6. Conclusion .................................................... 73
   6.1. Future Work .............................................. 73

A. CIC package coupling list .................................... 75

B. All objects view of JHotDraw 7 by JProfiler ................. 79

C. Allocation Hot Spot view of JHotDraw 7 by JProfiler ....... 81
## List of Tables

2.1. Structure and elements of behaviour view ........................................ 8  
2.2. Behaviour view analysis VS Static analysis ....................................... 12  
3.1. Characteristics of ISO 9126-1 Quality Model .................................. 14  
3.2. C & K metrics ................................................................................. 16  
3.3. Adaptation metrics to software architecture ..................................... 16  
3.4. Robert C. Martin metrics ................................................................. 18  
3.5. Static Metris VS Dynamic Metrics .................................................. 24  
4.1. Component metrics VS Class metrics ................................................. 28  
4.2. Function notation list ........................................................................ 31  
4.3. Cyclomatic Complexity threshold .................................................... 34  
4.4. SCCM threshold \((0 < N < 6)\) ........................................................... 35  
4.5. SCCM threshold \((N \geq 6)\) ............................................................... 36  
4.6. H.C.M1 threshold .............................................................................. 37  
4.7. H.C.M2 threshold .............................................................................. 37  
4.8. CW-H.C.M threshold ......................................................................... 38  
4.9. Basic idea for threshold integrity ....................................................... 39  
4.10. H.C.M1-SCCM threshold \((0 < N < 6)\) ............................................ 39  
4.11. H.C.M1-SCCM threshold \((N \geq 6)\) ................................................. 40  
4.12. H.C.M2-SCCM threshold \((0 < N < 6)\) ............................................ 41  
4.13. H.C.M2-SCCM threshold \((N \geq 6)\) ................................................. 42  
4.14. Integrity threshold ........................................................................... 42  
4.15. Implicit violations’ weights ............................................................. 52  
4.16. Explicit violations’ weights .............................................................. 53  
5.1. #.classes at runtime vs real .classes ................................................ 60  
5.2. Basic data of CIC for component metrics .......................................... 62  
5.3. Metrics’ results of CIC packages ....................................................... 62  
5.4. Final results of CIC packages .......................................................... 62  
5.5. validation-\(\rightarrow\)rules .................................................................... 64  
5.6. Comparing the results of four situations Situation Original: The original  
    package structure without any changes. Situation Facade 1: Design a  
    facade class in rules/monitoring. Situation Facade 2: Based on Facade 1,  
    optimize the methods in the facade class by aggregating methods which  
    were called at the same time. Situation Merging : Combine rules and  
    validation, marshalling and monitoring into new components. ............. 67
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.7. Basic data of JHotDraw 7 for component metrics</td>
<td>69</td>
</tr>
<tr>
<td>5.8. Metrics’ results of JHotDraw 7</td>
<td>70</td>
</tr>
<tr>
<td>5.9. Final results of JHotDraw 7 packages</td>
<td>70</td>
</tr>
<tr>
<td>E.1. Basic information table</td>
<td>85</td>
</tr>
<tr>
<td>E.2. relation table</td>
<td>85</td>
</tr>
</tbody>
</table>
List of Figures

2.1. hierarchy of views ......................................................... 7
2.2. Inheritance ............................................................... 9
2.3. Interface realization ..................................................... 10
2.4. Interface ................................................................. 10
2.5. Actual dependency relations in behaviour view ..................... 11

3.1. Sub-characteristics of ISO 9126-1 Quality Model .................. 14
3.2. Adapted characteristics for software architecture level .......... 15
3.3. Package level graph ..................................................... 19
3.4. Logical architecture design in Sonargraph ......................... 20
3.5. A sonargraph architect dashboard .................................. 20
3.6. Package dependency graph ........................................... 21
3.7. Details of a selected dependency edge ............................... 21

4.1. High coupling but low cohesion structure and loose coupling but high cohesion structure ..................................................... 27
4.2. An example for explaining definitions ................................. 31
4.3. Simple relations in a component ...................................... 35
4.4. H.C.M1-SCCM PLOT \(0 < N < 6\) ..................................... 40
4.5. H.C.M1-SCCM PLOT \(N \geq 6\) .......................................... 41
4.6. Situation 1 : methods invoked between components ............... 43
4.7. Situation 1 : methods invoked in a component ..................... 44
4.8. Situations of class hot spot ............................................ 45
4.9. Violation caused by inheritance ....................................... 47
4.10. Violation types based on principles ................................ 50
4.11. Examples of explicit rules ........................................... 50
4.12. Benefit to differentiate between explicit and/or implicit rules 51
4.13. Unimplemented relation of explicit violations ..................... 52
4.14. Result list of Simple violation weighted value metric .......... 54
4.15. Violations after checking based on planned architecture ....... 55
4.16. Results measured by two metrics .................................. 55

5.1. Evaluation workflow .................................................... 58
5.2. Simplified class diagram of the CIC class library (UML) .......... 59
5.3. Class diagram of the CIC drawn based on JSON files ............. 61
5.4. Left, components with direct relationships; right, the same components with a facade ........................................... 63
5.5. JHotDraw 7 package diagram ........................................... 68
A.1. CIC package coupling list 1 of 3 .................................... 76
A.2. CIC package coupling list 2 of 3 .................................... 77
A.3. CIC package coupling list 3 of 3 .................................... 78
B.1. Packages and classes of JHotDraw 7 monitored by JProfiler ................ 80
C.1. Allocation Hot Spot of JHotDraw 7 monitored by JProfiler ................ 82
D.1. CPU Hot Spot of JHotDraw 7 monitored by JProfiler .................. 84
## List of Source Codes

<table>
<thead>
<tr>
<th>Section</th>
<th>Code Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Person.java</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Student.java</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Difference.java</td>
<td>10</td>
</tr>
<tr>
<td>4.1</td>
<td>Apptest.java</td>
<td>48</td>
</tr>
<tr>
<td>4.2</td>
<td>ReflectApp.java</td>
<td>48</td>
</tr>
<tr>
<td>5.1</td>
<td>A data set of JSON files</td>
<td>59</td>
</tr>
<tr>
<td>5.2</td>
<td>A part of BezierPath.java</td>
<td>71</td>
</tr>
</tbody>
</table>
1. Introduction

We must accept finite disappointment, but we must never lose infinite hope.

*Martin Luther King*

The quality of software is a big issue today, and as well known, software architecture (SA) evaluation is a way which predicts the quality of a software product from a higher-level design description [DN02]. Actually, software architecture plays an important role to make software system development successful, it has large influence on the software qualities of software systems [Ben98a]. Because Software Architecture is a promising approach to bridging the gap between requirements and implementations [MKG99]. A good software architecture should meet the requirements of customers, as well as being a guideline of organizing the implementation phase well.

Software architecture evaluation is defined as 'a technique or method which determines the properties, strengths and weaknesses of software architecture or software architectural style or a design pattern' [SS12]. Now there are many methods to evaluate the quality of software architecture, basically, they are classified into two main categories: Early software architecture evaluation methods which are applied before implementation, and late software architecture evaluation methods which are applied after implementation [CKK]. Scenario-based Software Architecture Evaluation Methods are the main way of the early software architecture evaluation. So far there are many scenario-based methods such as Scenario-based Architecture Analysis Method (SAAM) [KBWA94], Architecture based Tradeoff Analysis Method (ATAM) [KKC00], Architecture-Level Modifiability Analysis (ALMA) [BLBvV04] and some extension methods like ESAAMI (Extending SAAM by Integration in the Domain) [Mol99], SAAMCS (SAAM for Complex Scenarios) [LRvV99]. In [SS12], the authors had compared and analysed these scenario-based methods. In this master thesis, I did not focus on the early software architecture evaluation methods, because these scenario-based evaluation methods are implemented without code, but my goal is to analyse the quality of software architecture at runtime, the code is necessary. Therefore, what I researched is the late software architecture evaluation methods. Software architecture Metrics are the main way to evaluate the quality of software architecture after implementation. Software architecture metrics allow us to quantitatively define the degree of success or failure of a software architecture and help architects make decisions to restructure the software architecture. There exist many metrics for measuring software architecture in static view, while neglecting the behaviour view of software architecture. Indeed, the behaviour is the one that actually supporting the execution of the various software use...
1. Introduction

cases. So measuring the quality of SA behaviour is very important as well.

My thesis goal is to find or adapt existing metrics which support for measuring the behaviour of software architectures, and if there is no such metrics, a metric proposal should be made and evaluated. Specifically, I focus on Object Oriented design architecture, and the main language is JAVA.

The thesis is organized as follows: In the **Background** part, some important items had been defined, I introduced the static view and behaviour view of software architecture, and I compared them to illustrate their difference, then the motivation of my master thesis was given. A survey and research were given in the **Related work**, which showed if there existed metrics that can be used or adapted to measure the quality of software architecture in the behaviour view. In the **Concept** part, I defined some new metric proposals based on issues which were found in these existing metrics. In the **Evaluation** part, two cases (CommunicationIntegrityCheck and JHotDraw 7) were evaluated by some of these new metrics, then, some of metrics in Sonargraph-Architect and JProfiler were used to measure the quality of their architectures as well. Finally, I compared these results and validated my new metric proposals. In the **Conclusion** part, I summarized my master thesis and described the further work of these new metric proposals.
2. Background

Genius only means hard-working all one's life.

MENDELEYEV

Contents

2.1. Terminology ..................................................... 3
2.2. 4+1 view model .................................................. 4
2.3. Static view of software architecture ......................... 5
2.4. Behaviour view of software architecture ..................... 7
2.5. Behaviour view VS Static view ................................ 8
2.6. Motivation ....................................................... 12

2.1. Terminology

Some basic terms related to software architecture evaluation and which are significant for this thesis are listed as following.

Software Architecture

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them" [BCK03]. Software Architecture defines software elements and the relation between those elements. These elements can be classes, components, interfaces and so on.

Software Architecture Description

A Software Architecture Description is not just an artful description of design but has far more depth to it. It is a set of descriptive models grouped into views. Each view emphasizes certain architectural aspects that are useful to different stakeholders and for different purposes [Kru95].

Software Component

Councill and Heineman [CH01] defined software component as 'a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard'. Szyperski et al [Szy02]
2. Background

introduced that "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties". In summary, a software component is a unit of software systems, which can be developed and deployed independently. While in my master thesis, one package or a set of packages which are responsible together for a certain task, was considered as a 'component', actually it was not a real software component because it didn’t adhere to these definitions, they are not separately deployable. The reason I considered them as components was that there was no architecture descriptions about components in the target software architecture, then I defined the 'component' based on package structure simply.

Metric

In IEEE-Glossar, metric is defined as "A quantitative measure of the degree to which an item possesses a given quality attribute". Quality metric is defined as ' A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute'.

2.2. 4+1 view model

The famous architecture view is 4+1 view model which was defined by Kruchten [Kru95]. It includes the logical view, the process view, the physical view, the development view, and the fifth view which is constituted by organizing the description of architectural decisions around the four views and illustrating them with a few selected use cases, or scenarios. The logic view describes the design’s object model when an object oriented design method is used. It is represented by class diagrams and templates. A class diagram contains a set of classes and their logic relationships. The process view describes the design’s concurrency and synchronization aspects. It can be used to estimate message flow and process loads. The physical view describes the mapping of the software onto the hardware and reflects its distributed aspect. So it impacts little on the quality of software architecture. The development view describes the organization of the actual software modules in the software development environment. It supports cost evaluation, planning, monitoring of project progress of the target software product. Designers represent the development view by module and subsystem diagrams which illustrate the system’s export and import relationships. Scenarios show how the elements of the four views work together seamlessly. The function of that is to help designers discover architectural elements, and validates the architecture design both on paper and the architectural prototype.
2.3. Static view of software architecture

2.3.1. Static analysis on source code

The 4+1 views can be represented by UML, and obviously these static view can be organized before the software implementation. Now there are various software metrics which have been created based on the source code analysis for measuring the quality of both the software product and software architecture. Static analysis on source code is not a new way to measure the quality of software systems. The basic idea is used by source code analysis tools to get the implementation static view. ISO 9126 as a well known standard for evaluating software quality, has defined 6 software quality characteristics in ISO 9126-1, and also gives metrics for each characteristics in ISO 9126-2 and ISO 9126-3. But there is no clear standard for software architecture evaluation. More details was discussed in Related work.

Evaluating the quality of a software architecture from the source code is a way to help software re-engineering actually. We use the data extracted from the source code to represent the implementing software architecture. The source code can reflect the actual relations between classes, components and layers. We call it static analysis on the source code because the measurements are not taken at runtime. The importance of coupling and cohesion in software architecture and programming is well understood [HM95]. Architects think that designs with low coupling and high cohesion lead to products that are both more reliable and more easily understandable. Stevens et al. [SMC74] first defined coupling as "the measure of the strength of association established by a connection from one module to another". In simple words, coupling means how well two software architecture elements are related, i.e how independent they are. These elements can be classes and components. Measuring coupling actually is measuring the dependency. In the source code, there are 4 types of coupling [OAS08]:

1. Parameter coupling, it is any method call, possibly including parameters.
2. External/file coupling, it refers to classes that access the same external medium, including external files.
3. Inheritance coupling, it occurs when one class is a subclass or descendant of another.
4. Global coupling, it refers to variables that are defined in one class and used in others.

For the parameter coupling, Java allows two explicit types of method calls, instance and static, and one implicit type, through a constructor.

- Instance call b.m(); explicit. If a method in class A explicitly calls method m() in class B through an object instance.

- Static call B.m(); explicit. Class A calls a public static method m() in class B.
2. Background

- Constructor call `B b = new B();` implicit. When a variable of type B is defined and instantiated in Class A.

Now there are many metrics for measuring the couplings for different entities, Coupling between object classes (CBO) is one of coupling metrics counting the number of other classes which is coupled to the target class [CK94]. Cohesion is defined as "a measure of the degree to which the elements of a module belong together" [SMC74]. In a highly cohesive component, all elements are related to the performance of a single function. [YC79] proposed six categories of module cohesion. They are Coincidental, Logical, Temporal, Procedural, Communicational, Sequential, and Functional. These categories are defined by an ordinal scale ranging from weakest (Coincidental) to strongest (Functional). Now most cohesion metrics focus on class cohesion. The basic one is Lack of Cohesion in Methods (LCOM) introduced in [CK94]. It is a metric based on class level. After that, some new cohesion metrics adapted from LCOM have been created, while most cohesion metrics are based on class level, few metrics measure the component cohesion. I will illustrate these existed metrics in Related work.

Complexity is another important factor for evaluating the quality of software systems and software architecture. It has been shown to be one of the major contributing factors to the cost of developing and maintaining software [Gra92]. It is hard to define the complexity independently. Joseph et al. [KST+86] defined complexity as "the difficulty of performing tasks such as coding, debugging, testing, or modifying the software". Zuse [DSD95] defined that 'The true meaning of software complexity is the difficulty to maintain, change and understand software. It deals with the psychological complexity of programs'. Software complexity measurement provides a quantitative method for predicting how difficult it will be to design, implement, and maintain the system. No matter measuring the method complexity or system complexity, they are both based on the coupling (incoming coupling and outgoing coupling) of measured entities mostly.

Architecture compliance check is also an important measurement based on source code. Because no matter how much effort is spent on preparing the architecture, there always exits some difference between the resulting implementation and designed architecture [RLGBAB08]. The compliance of the architecture can be checked statically (without executing the code) and dynamically (at runtime) [KP07]. Knodel et al. [KP07] identified three main static architecture compliance checking approaches:

- **Reflexion models**: The basic idea is to compare the architectural model (the planned or intended architecture) with a source code model (the actual or implemented architecture). It is a methodology.

- **Relation Conformance Rules**: Relation conformance rules specify the allowed or forbidden relations between components, then the actual relations in the source code will be checked to make sure if the rules are complied.

- **Component Access Rules**: These rules specify simple ports for components, which other components are allowed to call. Comparing with relation conformance rules, component access rules are more specific. These rules help to increase the information hiding of components on a level.
In summary, coupling, cohesion and complexity are three important factors which indicate the quality of software architecture, and can be measured in static analysis based on source code. There have existed some tools for measuring the quality of software architecture based on source code. Architecture compliance measurement is a way to check the difference between designed architecture and implemented architecture.

2.4. Behaviour view of software architecture

2.4.1. Definition

Maher Salah and Spiros Mancoridis [SM04] introduced a hierarchy of dynamic software views. It includes Object-interaction view, Class-interaction view, Feature-interaction view and Feature implementation view. Object-interaction view is obtained and organized based on the execution traces of the programs. It serves as the basis for higher level views. "Class-interaction view is an abstraction of the object-interaction view, where sets of objects are represented by their corresponding classes." Feature-interaction view illustrates the interactions between program features. Features are defined by users in terms of marked-traces. A marked trace is established manually during the execution of the program by specifying the start and the end of the trace. Feature implementation view is a mapping between program features and the classes that implement these features. Figure 2.1 shows the hierarchy of views.

Figure 2.1.: hierarchy of views

Depending on these existing view definitions, I try to define the behaviour view. The behaviour view describes the actual actions of a software system at runtime. The basic elements of behaviour view are object creation and object interaction. These elements should be traced from the execution traces of the program. While our goal is to analyse the quality of software architecture in behaviour view, so the structure of the software architecture is the point, and objects are too microcosmic to be used in software architecture evaluation. Thus, we need to abstract the object level to class level, which means using the corresponding classes of objects to represent the communication in behaviour view. We can abstract the class level to component level as well, in other words, we use the corresponding components of classes to represent the communication in behaviour view.
2.4.2. Object interaction

For object interactions, we know it should depend on the message passing between objects. There are two main kinds of message passing. One is invoking variable directly, another is method invoked. Most messages are passed by method invoked. Table 2.1 shows the structure and elements of behaviour view. I classify the behaviour view into three levels: object level, class level and component level. In object level, we know some objects will be created at runtime, which can not happen in class level and component level. But in class level, if some objects are created, they must be created by method invoked. So in class level, we can know this kind of behaviour through tracing method execution. As well known, component-based development (CBS) becomes more and more popular, because it makes software systems low complexity and these components can be reused easily in many different systems [LJK+01]. Therefore, evaluating the quality of software architecture in component level is very useful.

<table>
<thead>
<tr>
<th>Object</th>
<th>Object interactions</th>
<th>Object created</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Invoking variable directly</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Methods invoked</td>
</tr>
<tr>
<td>Class</td>
<td>Class interactions</td>
<td>Invoking variable directly, of any object belonging to this class</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Methods invoked</td>
</tr>
<tr>
<td>Component</td>
<td>Component interactions</td>
<td>Invoking variable directly, of any object belonging to this component</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Methods invoked</td>
</tr>
</tbody>
</table>

Table 2.1.: Structure and elements of behaviour view

2.5. Behaviour view VS Static view

After comparing the behaviour view and static view, I found that there are some difference between them. I listed these difference in the following parts. Firstly, the dependencies relations can be measured in behaviour view and static view, but the type of measured dependencies are different. In the static view, there are two key words "extends" and 'implements' in the source code to show the inheritance and interface implementation dependency relations. For example, we have two classes. Defining class Person in source code 2.1 and class Student in source code 2.2 class Student inherits from class Person.

```java
1 public class Person {
2   String name;
3   char sex;
4   Person(String n, char s){
5     name = n; sex = s;
6   }
7   public String getName() { 
```
2.5. Behaviour view VS Static view

```
8     return name;
9 }
10 public char getSex()
11     return sex;
12 }
13 }

Source Code 2.1: Person.java

1 public class Student extends Person{
2     long numS;
3     Date time;
4     Student(String n, char s, long nu, Date t) {
5         super(n, s);
6         numS = nu; time = t;
7         // TODO Auto-generated constructor stub
8     }
9     public long getNumber() {
10         return numS;
11     }
12     public Date getDate() {
13         return time;
14     }
15 }

Source Code 2.2: Student.java

We use static source code tools to measure the Java code and get the dependency graph like figure 2.2. But at runtime, if there is no method invoked between two object classes, the dependency between these two classes cannot be measured. That means the inheritance cannot be measured in behaviour view.

![Figure 2.2.: Inheritance](image)

The same situation is also found in interface implementation. An interface defines a set of methods, these methods should be realized in other classes like the figure 2.3. While at runtime, the interface implementation relations cannot be determined.
Another situation about interface is like the figure 2.4. class Difference calls the method of interface Calculation. The Java code is source code 2.3.

```java
public class Difference {

    float d;
    getDifferenceofArea();
}
```
public float d;
float getDifference()
{
    Calculation c = new Cycle();
    Calculation r = new Rectangle();
    float d = c.getArea() - r.getArea();
    return d;
}

Source Code 2.3: Difference.java

In class Difference, the class Cycle and class Rectangle are instantiated through interface Calculation. Actually, if these code are executed, the dependency relations should be figure 2.5 based on method track.

Figure 2.5.: Actual dependency relations in behaviour view

So in the behaviour view, if Class A depends on Class B, the situations are:

- Class A calls a method in Class B;
- Class A refers to a data member in Class B. It means Class A accesses a parameter of Class B directly.

There are two types of data that can only be measured in the behaviour view at runtime, method execution frequency and execution time. Method execution frequency means
how many times a method executed during runtime. Method execution time means how long a method executed consumed. The two particular attributes measured at runtime are important to be used in software architecture evaluation in the behaviour. Because they can reflect the actual characteristics of software architecture at runtime. Based on above analysis between static view and behaviour view, I summarize the difference in the table[2.2]. Because of the difference between static view and behaviour, if considering the behaviour view to measure the quality of software architecture, we can know more details about the architecture at runtime (method execution frequency and time consumed), know the difference of the results achieved from static view and behaviour view. Finally, we can combine the two measurement ways to make the evaluation more accurate.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Behaviour View</th>
<th>Static view</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Dependencies</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Interface Implementation</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Execution Frequency</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Execution Time</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 2.2.: Behaviour view analysis VS Static analysis

2.6. Motivation

The architecture of a software application or a family of software applications has a big impact on various vital aspects of the software, such as the malleability, maintainability, understandability, etc. Creating good architectures usually comes at the price of significant financial investments. This is why ensuring a good quality of the software architecture throughout the entire software project is of utmost importance. There have existed various metrics for measuring static view of software architecture, however, only a few metrics focus on measuring the behaviour view of software architecture at runtime. After comparing the static view with behaviour view of software architecture, I found some difference which has listed above. Therefore, we can get more details by measuring the software architecture in the behaviour view, and evaluate the quality of the software architecture at runtime. Maybe we can get different results and compare these results with the ones from the static analysis to check if they are better. Therefore, the metrics for measuring the behaviour view of software architecture are necessary. After researching these existing metrics, I found there is no such metrics, at least no suitable metrics for measuring the quality of software architecture in behaviour view. Then, I proposed three new kinds of metrics based on some existed static and dynamic metrics, and used them to measure and evaluated the quality of software architecture at runtime.
3. Related work

ISO/IEC 9126 is currently one of the most widespread quality standards for measuring the quality of software products. It defined quality as "a set of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs". Six independent quality characteristics have been proposed as a set of attributes of a software product, which can be used to describe and evaluate the quality of a software product. These characteristics and their descriptions are given in table 3.1.
3. Related work

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality</td>
<td>The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions</td>
</tr>
<tr>
<td>Reliability</td>
<td>The capability of the software product to maintain its level of performance under stated conditions for a stated period of time</td>
</tr>
<tr>
<td>Usability</td>
<td>The capability of the software product to be understood, learned, used and attractive to user, when used under specified conditions</td>
</tr>
<tr>
<td>Efficiency</td>
<td>The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions</td>
</tr>
<tr>
<td>Maintainability</td>
<td>The capability of the software product to be modified. Modifications may include corrections, improvements or adaptations of the software to changes in the environment and in the requirements and functional specifications</td>
</tr>
<tr>
<td>Portability</td>
<td>The capability of the software product to be transferred from one environment to another. The environment may include organizational, hardware or software environment</td>
</tr>
</tbody>
</table>

Table 3.1.: Characteristics of ISO 9126-1 Quality Model

For each characteristics, ISO 9126-1 defined a set of sub-characteristics to support them. Figure 3.1 shows the relations between characteristics and sub-characteristics.

Figure 3.1.: Sub-characteristics of ISO 9126-1 Quality Model
3.2. Static metrics

Losavio et al. [LCLRC03] adapted these quality characteristics of ISO 9126-1 to software architecture quality characteristics. Authors thought that software architecture as an intermediate product of the software development process, must satisfy the six ISO 9126 characteristics or a subset of them, so that it can help the final software products meet the quality model. Therefore, authors analysed every sub-characteristic and associated it with architectural design requirements. Finally, they got the set of characteristics for software architecture quality like figure 3.2 shows. Notice that there are only 5 characteristics, because authors thought *Usability* was related to the users. So all its sub-characteristics are independent from the architecture. *Maintainability* only contains two sub-characteristics, Coupling and Modularity. Coupling is a global property of the architecture and a system attribute. Modularity expresses the topology of the architecture.

![Figure 3.2.: Adapted characteristics for software architecture level](image)

3.2. Static metrics

"Static metrics which are collected by measurements made of the system representations such as the design, program or documentation. [Som01]." The static metrics are able to quantify various aspects of the complexity of design or source code of a software system based on the source code analysis without executing the software systems. They help to assess the complexity, understandability and maintainability of software system. These metrics can be classified into three different categories depending on their types. There are coupling metrics, cohesion metrics and complexity metrics. For static metrics, Chidamber and Kemerer [CK94] defined a set of metrics covering three categories for Object Oriented design. I summarize these metrics in the table 3.2.
### Related work

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weighted methods per class (WMC)</td>
<td>Measure the complexity and the number of methods within a class</td>
<td>Complexity</td>
</tr>
<tr>
<td>Depth of inheritance (DIT)</td>
<td>Measure how many layers of inheritance make up a target class hierarchy</td>
<td>Complexity</td>
</tr>
<tr>
<td>Number of children (NOC)</td>
<td>Measure the number of immediate subordinated to a class in the class hierarchy</td>
<td>Complexity</td>
</tr>
<tr>
<td>Response for a class (RFC)</td>
<td>Measure the set of all methods that can be invoked in response to a message to an object of the class</td>
<td>Complexity</td>
</tr>
<tr>
<td>Coupling between object classes (CBO)</td>
<td>Measure the number of other classes to which a target class is coupled</td>
<td>Coupling</td>
</tr>
<tr>
<td>Lack of cohesion in Methods (LCOM)</td>
<td>Measure the number of different methods within a class that reference a given instance variable</td>
<td>Cohesion</td>
</tr>
</tbody>
</table>

Table 3.2.: C & K metrics

Bengtsson [Ben98b] adapted C & K metrics [CK94] and a subset metrics proposed by Li and Henry [LH93] to measure the maintainability of software architecture. The redefined metrics are in table 3.3.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of methods (NOM)</td>
<td>Measure the number of methods of the architecture element’s interface</td>
</tr>
<tr>
<td>Number of abstract data types in a class (DAC)</td>
<td>Measure the number of place holders for architecture elements used to parametrize the element</td>
</tr>
<tr>
<td>Message Passing Coupling (MPC)</td>
<td>Measure the total number of outgoing messages from an element calculated from all message traces describing the architecture</td>
</tr>
<tr>
<td>Depth in Inheritance Tree (DIT)</td>
<td>measure the number of element types the element implements</td>
</tr>
<tr>
<td>Number of sub-Classes (NOC)</td>
<td>measure the number of elements implementing the element types</td>
</tr>
<tr>
<td>Response For a Class (RFC)</td>
<td>measure the number of methods available from the elements related to the element</td>
</tr>
<tr>
<td>Lack of Cohesion of Methods (LCOM)</td>
<td>the same with C &amp; K definition</td>
</tr>
</tbody>
</table>

Table 3.3.: Adaptation metrics to software architecture

LCOM is a basic metric for measuring cohesion, and many cohesion metrics are defined based on it, so I give more details on it. Suppose a class contains \( n \) methods, \( m_1, \ldots, m_n \),
3.2. Static metrics

and let \( \{I_i\} \) be the set of instance variables referenced by method \( m_i \). Two disjoint sets are defined as:

\[
P = \{(I_i, I_j) | (I_i \cap I_j) = \emptyset\}
\]

\[
Q = \{(I_i, I_j) | (I_i \cap I_j) \neq \emptyset\}
\]

Where:

\( P \) is the number of pairs of methods having no common instance variables,
\( Q \) is the number of pairs of methods having common instance variables. Then \( LCOM \) is defined as:

\[
LCOM = \begin{cases} 
|P| - |Q| & \text{if } |P| > |Q|, \\
0 & \text{otherwise.}
\end{cases}
\]

0 means the target class is cohesive because the number of pair methods which have common instance variables is more than the number of pair methods which have no common instance variables. There are some cohesion metrics adapted from it. Hitz and Montazeri [HM95] defined \( LCOM_3 \) and \( LCOM_4 \) to measure class cohesion. These two metrics measure the number of connected components of a graph. Bieman and Kang [BK95] defined Tight Class Cohesion (TCC) and Loose Class Cohesion (LCC). TCC is the relative number of directly connected methods. LCC is the relative number of directly or indirectly connected methods. Henderson-Sellers [Sel96] defined \( LCOM_5 \) which counts for each attribute how many methods access the attribute.

Hitz and Montazeri [HM95] defined coupling metrics in object level and class level, object level coupling (OLC) and class level coupling (CLC). OLC measures the number of objects to which the given object is coupled. CLC measures the number of classes to which the given class is coupled.

Cyclomatic Complexity (CC) is one of the most widely used and accepted software metrics. It was introduced by McCabe [McC76]. It is used to measure the number of linearly independent paths through a program module and it is a count of the number of test cases that are needed to unit-test the program module methodically. Cyclomatic Complexity is based on a connected graph of the program module. The formula is defined as:

\[
CC = E - N + 2P
\]

Where:

\( E \) is the number of edges of the graph,
\( N \) is the number of nodes,
\( P \) is the number of connected components.

Because of the growing popularity of Component Based Software Development (CBSD), some metrics are defined to assess the reusability, adaptability, composeability and flexibility of software components. Jungmayr [Jun02] defined the Average component dependency (ACD) to measure the component coupling. The formula is:
3. Related work

\[ ACD = \frac{1}{n} \times \sum_{i=1}^{n} CD_i \]

Where

\( n \) is the number of components in the system,

\( CD_i \) is the component dependency of component\(_i\), the number of components the component\(_i\) depends on directly and indirectly.

Robert C. Martin [Mar94] described a set of Object Oriented design quality metrics. These metrics measure the interdependence between the subsystems of the design. Designs which are highly interdependent tend to be rigid, unreusable and hard to maintain. He defined 'Class Category' as a group of classes which are highly cohesive. Later he used these metrics to measure the component design. I list these metrics in table 3.4.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description/Formula</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ca (Afferent Couplings)</td>
<td>The number of classes outside this category that depend upon classes within this category</td>
<td>Incoming couplings</td>
</tr>
<tr>
<td>Ce (Efferent Couplings)</td>
<td>The number of classes inside this category that depend upon classes outside this category</td>
<td>Outgoing couplings</td>
</tr>
<tr>
<td>I (Instability)</td>
<td>( Ce \div (Ca + Ce) )</td>
<td>This metric has the range ([0,1]). ( I = 0 ) indicates a maximally stable category. ( I = 1 ) indicates a maximally unstable category.</td>
</tr>
<tr>
<td>A (Abstractness)</td>
<td>( A = \frac{# \text{. abstract classes in category}}{# \text{. classes in category}} )</td>
<td>This metric range is ([0,1]). 0 means concrete and 1 means completely abstract. Interfaces are counted as abstract classes.</td>
</tr>
<tr>
<td>D (Distance)</td>
<td>(</td>
<td>(A + I - 1) \div 2</td>
</tr>
</tbody>
</table>

Table 3.4.: Robert C.Martin metrics

Kharb and Singh [KS08] introduced a suit of complexity metrics for components, they thought that the system complexity should be evaluated using three constituents of component integration:

- Complexity of incoming interactions
Saraiva et al. [SSC10] introduced three layered architecture design principles as following:

- **Back-Call Principle**: A top layer should depend only on a lower layer, but lower layers should not depend on the upper layers;
- **Skip-Call Principle**: Each layer should depend only on the layer located immediately below it;
- **Cyclic-Call Principle**: Cycles of dependencies between layers should not exist because they make a set of layers monolithic and inseparable.

Based on these principles, Archie Meter, which is a tool that supports developments in understanding how the implementation of an aspect-oriented system conforms to its intended layered software architecture, collects three simple metrics to measure the violations. They are:

- **Skip Call Violation (SCV)**: counting the number of skip call violations.
- **Back Call Violation (BCV)**: counting the number of back call violations.
- **Cyclic Call Violation (CCV)**: counting the number of cyclic call violations.

### 3.3. Static analysis tools

**Sonargraph-Architect** is a famous and widely used tool for software architecture evaluation in static view. It allows users to visualize and analyse the structure of any software system written in Java within minutes, and helps users to uncover unwanted and cyclic dependencies on all levels of architecture. Figure 3.3 shows the dependency graph in package level. From this graph, **Sonargraph-Architecture** can detect the number of cycles among packages and the size of every cycle.
3. Related work

Users can also define the logical architecture which is a set of rules designating allowed and forbidden dependencies in designed architecture, and then check them in source code. Figure 3.4 shows the logical architecture design in this tool. These rules would be "the UI layer is not allowed to directly access the DAO layer" or "the UI layer cannot use JDBC directly". Violations will be found and marked with error markers.

**Sonargraph Architect** calculates some metrics and monitors them to give users an early warning if things move into the wrong direction. For example, the metric Average Component Dependency (ACD) would tell you, on average, how many Java files would be affected by a change in your system. Actually, ACD is a coupling metric. After measurements, the dashboards show all of the key metrics in one place so that users can have an immediate overview of the overall technical quality of a software architecture. Figure 3.5 is a dashboard which illustrates all results of selected metrics. The metric thresholds are monitored by users.
3.3. Static analysis tools

STAN is another famous structure analysis tool based on static code analysis. It presents and visualizes the structural design in a way that is easy to understand for users based on static code. The dependency graphs for all levels of abstraction (class level, package level) can be given. Figure 3.6 shows a package level dependency graph. When users select a edge in the dependency graph, more details including source name, type of relation, and target name will be illustrated like figure 3.7 shows. The dependency graphs include composition view, coupling view (incoming and outgoing dependencies), and sandbox view which allows to look at dependencies between any classes, packages, package trees or libraries.

![Dependency Graph](image)

Figure 3.6.: Package dependency graph

![Dependency Details](image)

Figure 3.7.: Details of a selected dependency edge

Based on these details and dependency relations, STAN computes several metrics to measure the quality of structure design. These metrics are:

- Several counting metrics
- Estimated Lines of Code
- McCabe’s Cyclomatic Complexity
3. Related work

- Average Component Dependency, Fat and Tangled
- Metrics by Robert C. Martin
- Metrics by Chidamber & Kemerer

Cycles can be detected as well by STAN. After discovering the cycles, a naive approach will be given to try to break them. Finally, STAN gives a report about the target structural design.

In summary, these static analysis tools are based on visualizing the source code and measuring the dependencies to evaluate the quality of software architecture. But the problem is that there is no clear standards for what a good software architecture is. These thresholds for metrics are not explicit, they are all monitored by users.

3.4. Dynamic metrics

"Dynamic metrics which are collected by measurements made of a program in execution.[Som01]." Dynamic metrics are the class of software metrics that capture the dynamic behaviour of the software system. The data of dynamic metrics are usually obtained from the execution traces of the code or from the executable models. The same with static metrics, most dynamic metrics aims to measure the coupling, cohesion and complexity of software products or software architectures. They help to assess the efficiency and the reliability of a program.

Dynamic coupling metrics are used to measure actual coupling taking place between objects. So that means dynamic coupling metrics are measured at object level, but they can be aggregated to class or component level. Yacoub et al. [YAR99] introduced object level dynamic coupling metrics, Export Object Coupling (EOC) and Import Object Coupling (IOC). $EOC_x(o_i, o_j)$ describes the export coupling for object $o_i$ with respect to object $o_j$. It is the percentage of the number of messages sent from $o_i$ to $o_j$ with respect to the total number of messages exchanged during the execution of the scenario $x$. The formula is:

$$EOC_x(o_i, o_j) = \frac{|\{M_x(o_i, o_j)\} o_i, o_j| \in O \land o_i \neq o_j|}{MT_x} \times 100$$

Where:
- $M_x(o_i, o_j)$ is the number of messages sent from $o_i$ to $o_j$,
- $MT_x$ is the total number of messages exchanged during the execution of the scenario $x$.
- $IOC_x(o_i, o_j)$ describes the import coupling for object $o_i$ with respect to object $o_j$. It is the percentage of the number of messages received by object $o_i$ and was sent by object $o_j$ with respect to the total number of messages exchanged during the execution of the scenario $x$. The formula is:

$$IOC_x(o_i, o_j) = \frac{|\{M_x(o_j, o_i)\} o_i, o_j| \in O \land o_i \neq o_j|}{MT_x} \times 100$$
3.4. Dynamic metrics

Where:
\( M_x(o_j, o_i) \) is the number of messages received by object \( o_i \) from object \( o_j \),
\( M_{T_x} \) is the total number of messages exchanged during the execution of the scenario \( x \).

Arisholm [ABF04] extended the concept of import and export coupling given by Yacoub et al. to define 12 dynamic coupling measures for object-oriented software. These metrics are based on object level (six metrics) and class metrics (6 metrics). There are three parts in these metrics. First, direction of coupling, it includes import coupling (IC) and export coupling (EC). Secondly, the mapping level, it includes object level (O) and Class level (C). Finally is the strength of coupling, it includes dynamic messages (D), distinct methods (M) and distinct classes (C). For example, \( IC_{OD} \) is one of the 12 metrics, it measures the total number of dynamic messages sent from one object to other objects, it maps to object level and represents the import coupling of an object. So these metrics measure more details at different levels.

Mitchell and Power [MP04] defined two metrics based on C & K’s LCOM, Runtime Simple LCOM (\( R_{LCOM} \)) and Runtime Call-Weighted LCOM (\( RW_{LCOM} \)). \( R_{LCOM} \) considers instance variables that are actually accessed at runtime. \( RW_{LCOM} \) was defined by weighing each instance variable by the number of times it is accessed at runtime. That means \( RW_{LCOM} \) considers the execution frequency at runtime.

\[
P^W = \sum_{1 \leq i,j \leq n} \{(N_i + N_j)|I_i \cap I_j = \emptyset\}
\]
\[
Q^W = \sum_{1 \leq i,j \leq n} \{(N_i + N_j)|I_i \cap I_j \neq \emptyset\}
\]
\[
RW_{LCOM} = \begin{cases} 
|P^W| - |Q^W| & \text{if } |P^W| > |Q^W|, \\
0 & \text{otherwise.}
\end{cases}
\]

Where:
\( \{I_i\} \) is the set of instance variables used by method \( m_i \) at runtime.
\( \{I_j\} \) is the set of instance variables used by method \( m_j \) at runtime.
\( N_i \) is the number of times, method \( m_i \) dynamically accesses instance variables from the set \( \{I_i\} \).
\( N_j \) is the number of times, method \( m_j \) dynamically accesses instance variables from the set \( \{I_j\} \).

Cho et al. [CKK01] introduced a metrics suit to measure component quality. It proposed four kinds of complexity metrics to measure component complexity which are Component Plain Complexity (CPC), Component Static Complexity (CSC), Component Dynamic Complexity (CDC), and Component Cyclomatic Complexity (CCC). He described CDC as a metric that measures the complexity of internal message passing in a component with a dynamic view. In CDC, the message passed between classes, the frequency of messages passed between classes and the
3. Related work

Complexity of each message are used to measure the complexity of components. While these metrics are not validated.

Chhabra and Gupta [CG10] made a comparison between static and dynamic metrics. Table 3.5 shows the comparison.

<table>
<thead>
<tr>
<th>Static Metrics</th>
<th>Dynamic Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simpler to collect</td>
<td>Difficult to obtain</td>
</tr>
<tr>
<td>Available at the early stage of software development</td>
<td>Accessible very late in software development life-cycle</td>
</tr>
<tr>
<td>Less accurate than dynamic metrics in measuring qualitative attributes of software</td>
<td>Suitable for measuring quantitative as well as qualitative attributes of software</td>
</tr>
<tr>
<td>Deal with the structural aspects of the software system</td>
<td>Deal with the behavioural aspects of the system also</td>
</tr>
<tr>
<td>Inefficient to deal with dead code and OO features such as inheritance, polymorphism and dynamic binding</td>
<td>Dynamic metrics are capable to deal with all object-oriented features and dead code</td>
</tr>
<tr>
<td>Less precise than dynamic metrics for the real-life systems</td>
<td>More precise than static metrics for the real-life systems</td>
</tr>
</tbody>
</table>

While I’m not agree some of their points. Actually, the static metrics which are based on static source code analysis, can measure the inheritance and polymorphism, but in the behaviour view, the inheritance relations are not measured efficiently as I mentioned in Background. About which kind of metrics are more accurate, I think each kind of metrics has its own advantages, what we can do is to combine the both kinds of metrics to make the software evaluation more accurate.

3.5. Java profilers

Java profilers are used to analyse the performance of software programs at runtime. They can monitor memory usage and leak in any Java applications, monitor method call duration, and display information graphs, reports, tree views, etc. JProfiler [JPr] is a powerful Java profiler tool, the profiling views in JProfiler includes Memory profiling, CPU profiling, Thread profiling, Monitor profiling and so on. It is a powerful and useful tool for analysing the performance of Java applications.
4. Concept

Man can only be free through mastery of himself.

S.E.Morison

4.1. Problems with the existing metrics

After surveying these exiting metrics, I found that in the behaviour view, the method execution frequency and time consuming can be measured, but few metrics used these data to measure the software architecture. So we should consider these unique data to measure the quality of a software architecture in the behaviour view. There are some specific problems that have not been solved in the behaviour view. They are:

- Are the components well designed in the behaviour view? (Q1) Because there are few component metrics to measure the component cohesion and complexity at runtime.

- What method calls have slowed down the current execution? (Q2) This problem can be measured at runtime, and we need to reflect them in the architecture.

- What classes should be redesigned or recoded in order to optimize execution performance and architecture? (Q3)

- What violations should I address first? (Q4) A method for ranking the violations should be proposed, so that when restructuring the architecture, we can know where we should start.

Now there are not any clear standards showing what a software architecture is good. Generally, if the elements of a software system such as components are needed to be re-usable in other software systems, a software architecture design with loose couplings between components and high cohesion within components will be good. If a software system would be modified often or easy to maintain, a good architecture design should
4. Concept

be loose coupling and less complex. If a software system requires high performance, the time-based behaviour of elements and the frequency and volume of inter-element communication should be paid more attention in the architecture design. The existing methods, only evaluate the quality of software architecture with particular aspects such as reusability, maintainability or flexibility. Assessing the quality of software architecture in the behaviour view should be done after implementing the source code and executing the software systems. Therefore, the goal of measuring the architecture quality in the behaviour view is to help optimize or refactor the existing architectures. In my approaches, the proposals are towards the architecture refactoring. Lichter [LL12] has defined the metric development as six steps:

1. Identifying the relevant aspects.
2. Modelling the relevant aspects.
3. Defining the scale.
4. Define the formula.
5. Define the threshold.
6. Using and Calibrating the metric.

In the following proposal metrics’ descriptions, the steps 1 to 4 were used to define them.

4.2. Component metrics

Councill and Heineman [CH01] defined software component as "a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard". Component-based development (CBD) is an important and well-known software development method. It aims to compose a software system into some software units or components. In the architecture view, the architecture is made up with different components. The benefits of CBD is that increasing reusability, reducing production cost. So if evaluating the quality of software architecture, we can start with analysing the component design. For answering the question "Are the components well designed?", we need to ask further questions:

- What characteristics are important for component design?
- How to measure these attributes in the behaviour view?
4.2. Component metrics

4.2.1. Identifying the relevant aspects

Analysis of component design characteristics

As mentioned, CBD increases the reusability and reduces the production cost. Because software components can be reused to different software systems easily and more quickly than writing new code [LW07]. That means a component should be loosely coupled with other components. Considering the definition of software component, we can know that a component can be deployed and developed independently, and the basic idea of component design is to realize different main functions in different components independently. These components interact through interfaces to keep loose coupling. A software component encapsulates a set of related functions, so components are modular and cohesive. Based on the analysis, we can know that coupling and cohesion are two important indicators for component design.

Measuring coupling and cohesion in component design

Coupling and cohesion are normally interdependent, the higher the cohesion of individual building blocks in an architecture, the lower the coupling between the building blocks. They affect the software architecture flexibility, maintainability and so on. Figure 4.1 shows a structure graph with high coupling but low cohesion and a structure graph with loose coupling but high cohesion.

![Figure 4.1.: High coupling but low cohesion structure and loose coupling but high cohesion structure](image)

Obviously, the right structure graph is better than the left structure graph. Thus, if these blocks are components, the small rectangles are classes, a good component design should be that the relations in this components must be more than the coupling relations at least. There have existed some metrics for measuring coupling of class and component. The basic idea of class coupling metrics is to count the number of relations between the target class and other classes. The component coupling metrics are similar with class coupling metrics. They measure the number of relations between the target component and other components. For the class cohesion metrics, like LCOM3, LCOM4 and LCOM5 (introduced in Related work), they are based on LCOM, which based on the instance variables used by the methods in the target class. It is difficult to find some component metrics which measure the cohesion of a component exactly. Yu et al. [YLZD08] introduced an approach to measuring component cohesion based on
4. Concept

structure entropy. The basic idea is to divide the classes relations in a component into three parts:

- The relations between input-interfaces and operations (the normal classes);
- The relations among operations;
- The relations between operations and output-interfaces.

Then it defined three cohesion metrics for the three parts respectively based on structure entropy. The component cohesion is the sum of three parts’ cohesion with different weight $\alpha$, $\beta$ and $\gamma$, and $\alpha + \beta + \gamma = 1$. The final component cohesion value is from 0 to 1. These metrics are based on static code analysis. I make a comparison between component metrics and class metrics in coupling and cohesion in table.

<table>
<thead>
<tr>
<th>Coupling</th>
<th>Component</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>The number of</td>
<td>The number of relations between the target component and other components</td>
<td>The number of relations between the target class and other classes</td>
</tr>
<tr>
<td>Cohesion</td>
<td>Based on the classes relationships in the target component</td>
<td>Based on the instance variables used by the methods in the target class</td>
</tr>
</tbody>
</table>

Table 4.1.: Component metrics VS Class metrics

Therefore, measuring a component cohesion should depend on two aspects, structure cohesion and function cohesion.

4.2.2. Modelling the relevant aspects

Considering an software architecture including n components, a component k ($Com_k$) with n classes, $C_1, ..., C_n \subseteq Com_k$. Let $M_i$ be the set of methods which are invoked by $C_i$ (these methods are not defined in Class i), $M'_i$ be the set of methods defined in $C_i$, $A_i$ be the set of parameters which are accessed by $C_i$ (these parameters are not defined in $C_i$), $A'_i$ be the set of parameters which are defined in $C_i$.

$$P^R = \{(C_i, C_j)|M_i \cap M'_j = \emptyset \land M_j \cap M'_i = \emptyset \land A_i \cap A'_j = \emptyset \land A'_i \cap A_j = \emptyset, C_i, C_j \subset Com_k\}$$

Thus, $P^R$ contains all the pair of classes ($C_i, C_j$) that have no direct relation with each other. Notation: $R$ means measured at runtime. $|P^R|$ means the number of pair classes which have no relation, and ($C_i, C_j$) means there is no direct relation from class i to class j, as well as no direct relation from class j to class i. Therefore, ($C_i, C_j$) and ($C_j, C_i$) are the same elements, we only use one to represent them.

$$Q^R = \{(C_i, C_j)|M_i \cap M'_j \neq \emptyset \lor A_i \cap A'_j \neq \emptyset, C_i, C_j \subset Com_k\}$$
4.2. Component metrics

$Q^R$ contains all the pair of classes (($C_i, C_j$)) that have at least one direct relation with each other. Notation: From the definition of $Q^R$, we can know that ($C_i, C_j$) and ($C_j, C_i$) represents different elements. ($C_i, C_j$) means there are relations that $C_i$ invokes a method or accesses a parameter which is defined in $C_j$, $C_i$ is a caller and $C_j$ is a callee. But ($C_j, C_i$) means there are relations that $C_j$ invokes a method or accesses a parameter which is defined in $C_i$, $C_j$ is a caller and $C_i$ is a callee. Then, if $C_i, C_j$ belongs to $Q^R$, it does no imply that $C_j, C_i$ belongs to $Q^R$. $|Q^R|$ means the number of pair classes which has direct relations with each other.

$$Q^R_{c_i, c_j} = \{M_i \cap M'_j, M_j \cap M'_i, A_i \cap A'_j, A_j \cap A'_i, C_i, C_j \subset \text{Com}_k\}$$

$Q^R_{c_i, c_j}$ is the set of methods and parameters which make Classes $C_i$ and $C_j$ have direct relations. Notation, $C_i$ and $C_j$ are in a same component.

$$|Q^R_{c_i, c_j}| = |M_i \cap M'_j| + |M_j \cap M'_i| + |A_i \cap A'_j| + |A_j \cap A'_i|$$

$|Q^R_{c_i, c_j}|$ means the number of relations between classes $C_i$ and $C_j$. One method invoked or one parameter accessed between $C_i$ and $C_j$ are treated as a specific relation.

$$|Q^R_{\text{Com}_k}| = \sum_{1 \leq i, j \leq n} |Q^R_{c_i, c_j}|, (c_i, c_j \subset \text{Com}_k)$$

$|Q^R_{\text{Com}_k}|$ means the total number of relations in the Component $k$.

$$O^R = \{(c_i, c_j) : M_i \cap M'_j \neq \emptyset \lor M_j \cap M'_i \neq \emptyset \lor A_i \cap A'_j \neq \emptyset \lor A_j \cap A'_i \neq \emptyset, C_i \in \text{Com}_k, C_j \in \text{Com}_l, \text{Com}_k \neq \text{Com}_l\}$$

$O^R$ means a set of pair classes which are not in a same component has at least a direct relation.

$$O^R_{c_i, c_j} = \{M_i \cap M'_j, M_j \cap M'_i, A_i \cap A'_j, A_j \cap A'_i, C_i \in \text{Com}_k, C_j \in \text{Com}_l, \text{Com}_k \neq \text{Com}_l\}$$

$O^R_{c_i, c_j}$ is the set of methods and parameters which make Class $C_i$ and $C_j$ have direct relations. Notation, $C_i$ and $C_j$ are in different components.

$$|O^R_{c_i, c_j}| = |M_i \cap M'_j| + |M_j \cap M'_i| + |A_i \cap A'_j| + |A_j \cap A'_i|$$

$|O^R_{c_i, c_j}|$ means the number of relations between classes $C_i$ and $C_j$. One method invoked or one parameter accessed between $C_i$ and $C_j$ are treated as a specific relation.

$$|O^R_{\text{Com}_k}| = \sum_{1 \leq i, j \leq n} |O^R_{c_i, c_j}|, (c_i \in \text{Com}_k, c_j \notin \text{Com}_k)$$

$|O^R_{\text{Com}_k}|$ means the total number of couplings of Component $k$. In order to make the method and parameter execution frequency as factors, I define the $Ef$ (execution
frequency) to represent how many times a method is executed or a parameter is accessed. So for a method $m_i$, $Ef(m_i)$ means times which method $m_i$ is executed, for a parameter $A_i$, $Ef(m_i)$ represents the times which parameter $A_i$ is accessed. Then based on the definition of $Q_{c_i,c_j}^R$ and $O_{c_i,c_j}^R$, we assume methods $m_1, ..., m_n \subset Q_{c_i,c_j}^R$ and parameters $a_1, ..., a_k \subset Q_{c_i,c_j}^R$, methods $m_1, ..., m_l \subset O_{c_i,c_j}^R$ and parameters $A_1, ..., A_h \subset O_{c_i,c_j}^R$. Then we have:

$$|Q_{c_i,c_j}^R| = \sum_{m \in \text{methods}} Ef(m) + \sum_{a \in \text{parameters}} Ef(a)$$

$|Q_{c_i,c_j}^R|$ means the total number of methods executed and parameters accessed between classes $C_i$ and $C_j$, $C_i$ and $C_j$ are in the same component.

$$|Q_{Com_k}^R| = \sum_{1 \leq i,j \leq n} |Q_{c_i,c_j}^R|$$

$|Q_{Com_k}^R|$ means the total execution frequency of internal relations in component $k$.

$$|O_{c_i,c_j}^R| = \sum_{m \in \text{methods}} Ef(m) + \sum_{a \in \text{parameters}} Ef(a)$$

$|O_{c_i,c_j}^R|$ means the total number of methods executed and parameters accessed between classes $C_i$ and $C_j$, $C_i$ and $C_j$ are in different components, $C_i \in Com_k$ and $C_j \notin Com_k$.

$$|O_{Com_k}^R| = \sum_{1 \leq i,j \leq n} |O_{c_i,c_j}^R|$$

$|O_{Com_k}^R|$ means the total coupling execution frequency of component $k$. The following table 4.2 lists the function notations which are used in new metrics, and figure 4.2 shows an examples to calculate these function notations.
4.2. Component metrics

<table>
<thead>
<tr>
<th>Function Notation</th>
<th>Description</th>
<th>Alternating quantity</th>
<th>Range</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>Q^R</td>
<td>)</td>
<td>The number of pair classes having direct relations</td>
<td>( \geq 0 )</td>
</tr>
<tr>
<td>(</td>
<td>P^R</td>
<td>)</td>
<td>The number of pair classes having no direct relations</td>
<td>( \geq 0 )</td>
</tr>
<tr>
<td>(Q_{c_i,c_j}^R)</td>
<td>The set of methods and parameters between classes (C_i) and (C_j), where (C_i) is the caller, and (C_j) is the callee</td>
<td>( C_i, C_j )</td>
<td>( C_i, C_j \subset \text{Com}_k )</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>Q_{c_i,c_j}^R</td>
<td>)</td>
<td>The number of relations between classes (C_i) and (C_j)</td>
<td>( C_i, C_j )</td>
</tr>
<tr>
<td>(</td>
<td>Q_{\text{Com}_k}^R</td>
<td>)</td>
<td>The total number of relations in the Component (k), excluding the relations a class has with itself</td>
<td>( \text{Com}_k )</td>
</tr>
<tr>
<td>(</td>
<td>Q_{\text{Com}_k}^R</td>
<td>)</td>
<td>The total internal execution frequency of component (k)</td>
<td>( \text{Com}_k )</td>
</tr>
<tr>
<td>(O_{c_i,c_j}^R)</td>
<td>The set of methods and parameters between classes (C_i) and (C_j), where (C_i, C_j) are in different components</td>
<td>( C_i, C_j )</td>
<td>( \geq 0 )</td>
<td>( C_i \in \text{Com}_k, \ C_j \in \text{Com}_l, \ \text{Com}_k \neq \text{Com}_l )</td>
</tr>
<tr>
<td>(</td>
<td>O_{c_i,c_j}^R</td>
<td>)</td>
<td>The number of relations between classes (C_i) and (C_j), where (C_i, C_j) are in different components</td>
<td>( C_i, C_j )</td>
</tr>
<tr>
<td>(</td>
<td>O_{\text{Com}_k}^R</td>
<td>)</td>
<td>The total number of couplings of Component (k)</td>
<td>( \text{Com}_k )</td>
</tr>
<tr>
<td>(</td>
<td>O_{\text{Com}_k}^R</td>
<td>)</td>
<td>The total coupling execution frequency of component (k)</td>
<td>( \text{Com}_k )</td>
</tr>
</tbody>
</table>

Table 4.2.: Function notation list

![Diagram 4.2.: An example for explaining definitions](image-url)

Figure 4.2.: An example for explaining definitions
4. Concept

For Component A, the results are:

- \( P^R = \{(\text{Class1, Class4}), (\text{Class2, Class3})\}, |P^R| = 2 \)
- \( Q^R = \{(\text{Class1, Class3}), (\text{Class1, Class2}), (\text{Class2, Class4})\}, |Q^R| = 3 \)
- \( Q^R_{c_1, c_3} = \{\text{getRules()}, \text{getLayer()}\}, |Q^R_{c_1, c_3}| = 2 \)
- \( \ |Q^R_{c_1, c_3} | + |Q^R_{c_1, c_2} | + |Q^R_{c_2, c_4} | = 4 \)
- \( |Q^R_{Com_A}| = 50 + 4 + 5 + 15 = 74 \)
- \( O^R_{c_3, c_5} = \{\text{getCallerCallee()}\}, O^R_{c_4, c_6} = \{\text{getCaller()}, \text{getCallee()}\} \)
- \( |O^R_{c_3, c_5}| = 1, |O^R_{c_4, c_6}| = 2, |O^R_{Com_A}| = 2 + 1 = 3 \)
- \( |O^{R}_{Com_A}| = 100 + 20 + 20 = 140 \).

4.2.3. Define the scale

It is a good way to represent the degree of a metric’s result by using percentage values. It is easy to show how a result looks like. For the component metrics, I propose the scale of it from 0 to 1. 0 represents that there is not any relations between classes in a component, 1 means there is not any coupling relations between the target component and other components. High values represent good cohesion except value 1.

4.2.4. Define the formula

Simple Component Cohesion Metric (SCCM)

The first metric proposal is Simple Component Cohesion Metric (SCCM), it measures the component structure cohesion, the formula is

\[
SCCM = \frac{|Q^R|}{|Q^R| + |P^R|}
\]

SCCM measures the simple structure cohesion of a component. The range is from 0 to 1. 0 means there is no relation between any two classes in the target component. 1 means any two classes have at least one relation in the target component. Obviously, the bigger the value of SCCM is, the better the target component cohesion is.

Hybrid Component Cohesion Metric (H.C.M)

The second metric proposal is Hybrid Component Metric, it measures the component function cohesion, the formula is

\[
H.C.M_{1_{Com_k}} = \frac{|Q^R_{Com_k}|}{|Q^R_{Com_k}| + |P^R_{Com_k}| + |O^R_{Com_k}|}
\]
I combine the internal relations (cohesion) and coupling relations (coupling) of the target component to measure the component cohesion design. So I call it Hybrid Component Cohesion Metric. We know for a pair related classes, there may exist more than one relation caused by methods invoked or parameter accessed. Then, I consider all the relations of a pair related classes to measure a component cohesion design. Cohesion and Coupling are interdependency. So if a component is designed as high cohesion, that means the coupling of this component is low or loose. In other words, the number of internal relations in this component should be more than the number of couplings of this component. In the H.C.M1, if the $|Q'_{\text{Com}_k}|$ is bigger than $|O'_{\text{Com}_k}|$, the value will be high and can reflect the cohesion of target component. The H.C.M1’ range is from 0 to 1, 0 means there is no relation between any two classes in the target component, it is the worst situation. 1 means the target component has no relation with other components and is never used, it is also a bad situation. Therefore, from 0 to 1 (not equal 1), the higher the H.C.M is, the better the component hybrid cohesion is.

After using H.C.M to measure some components, I found a big problem. That is the H.C.M value is declining with the number of classes (N) in a component increasing. The main factor causing this case is the $|P_R|$. For example, if N=20, and the class diagram is a simple connected graph and $|Q_R|=19$, then $|P_R| = C^2_N - 19 = 171$, it is a big number which will influence the H.C.M value much. In order to avoid this case, I define another H.C.M as:

$$H.C.M2_{\text{Com}_k} = \frac{|Q_{\text{Com}_k}^R|}{|Q_{\text{Com}_k}^R| + |O_{\text{Com}_k}^R|}$$

H.C.M2 is decided only by two factors: $|Q_{\text{Com}_k}^R|$ and $|O_{\text{Com}_k}^R|$. The H.C.M2’ range is from 0 to 1, 0 means there is no relation between any two classes in the target component, it is the worst situation. 1 means the target component has no relation with other components and is never used, it is also a bad situation.

**Call-Weighted Hybrid Component Cohesion Metric (CW-H.C.M)**

The third metric proposal is Call-Weighted Hybrid Component Cohesion Metric, it measures a component function cohesion, the formula is

$$CW - H.C.M_{\text{Com}_k} = \frac{|Q_{\text{Com}_k}^{pR}|}{|Q_{\text{Com}_k}^{pR}| + |O_{\text{Com}_k}^{pR}|}$$

$CW - H.C.W$ is defined by weighing each method or parameter by the number of times it is executed or accessed. As mentioned, a component is a set of classes which aim to realize a same function. Then, at runtime, a cohesive component should have more internal execution frequency than coupling execution frequency. The CW-H.C.M’ range is from 0 to 1. 0 means the worst situation, no method caused internal relation is executed. 1 is also a terrible situation, because it means this component is never used. Generally, from 0 to 1 (not equal 1), the higher the value is, the better the component hybrid cohesion is.
4.2.5. Define the threshold

There is no explicit standard elaborating when a component is well designed. All the papers said a component with low coupling and high cohesion is good. Lippert and Roock \cite{LR06} described that there was a rule of thumb that can somewhat serve as a guideline for architecture smells: 'If an element consists of more than 30 sub-elements, it is highly probable that there is a serious problem.' Then they gave some suggest rules for software architecture design.

a) Methods should not have more than an average of 30 code lines.

b) A class should contain an average of less than 30 methods.

c) A package should not contain more than 30 classes.

d) Subsystems with more than 30 packages should be avoided.

Therefore, based on these rules, I propose that a small component should contain less than 30 classes, and a composite component should not be consisted of more than 30 small components. Thus, in my three component metrics proposals, the number of classes \(N\) in a component is \(0 < N \leq 30\) by default. Then, I try to define thresholds for my metric proposals through considering complexity and some assumptions to evaluate a component design.

**SCCM threshold**

Consider the complexity of a component design, I use McCabe Cyclomatic Complexity (CC) to measure a component complexity. The formula is:

\[
CC = e - n + 2p
\]

Where \(e\) means the number of edges, \(n\) means the number of nodes, \(p\) means number of disconnected parts of the graph.

We assume that a component includes \(N\) classes, then the edges \(e = |Q^R|\), nodes \(n = N\) and \(p=1\) (assume the component graph as a connected graph). We get:

\[
CC = |Q^R| - N + 2
\]

And many websites and papers show the CC threshold:

<table>
<thead>
<tr>
<th>Cyclomatic Complexity</th>
<th>Risk Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-10</td>
<td>A simple module without much risk</td>
</tr>
<tr>
<td>11-20</td>
<td>A more complex module with moderate risk</td>
</tr>
<tr>
<td>21-50</td>
<td>A complex module of high risk</td>
</tr>
<tr>
<td>51 and greater</td>
<td>An untestable program of very high risk</td>
</tr>
</tbody>
</table>

Table 4.3.: Cyclomatic Complexity threshold
4.2. Component metrics

We have $SCCM = |Q^R|/(|Q^R| + |P^R|)$, and $|Q^R| + |P^R| \geq C_n^2$ ($C_n^2$ means any two classes have a relation). Then

$$SCCM \leq \frac{|Q^R|}{C_n^2}$$

Equality holds if any two classes have only one relation. Using the CC threshold to define SCCM threshold, we get:

$$SCCM \leq C_n^2 - N + 2 \leq 10$$

$$SCCM \leq C_n^2 - N + 2 \leq 20$$

After simplification,

$$SCCM \leq (16 + 2N)/(N(N - 1))$$

$$SCCM \leq (36 + 2N)/(N(N - 1))$$

Considering a component with N classes, we assume the relations among classes like figure 4.3, N-1 is the fewest number of relations in this component to make component graph be a connected graph. Therefore, we get $(N - 1)/C_n^2$, after simplification, the result is $\frac{2}{N}$.

![Figure 4.3: Simple relations in a component](image)

The SCCM is from 0 to 1. We assume SCCM=1, then from the formula $SCCM \leq \frac{|Q^R|}{C_n^2}$, we get $0 \leq N \leq 6$. That means if a component includes classes fewer than 6, no matter how many relations among classes in this component, the CC is fewer than 10, so this component is a simple module without much risk. Hence, the bigger the value of SCCM for this component is, the higher the component cohesion is. Assuming $|Q^R| \geq 2|P^R|$ means high cohesion, we get $SCCM \geq 0.66$. Assuming $|Q^R| \leq |P^R|$ means low or bad cohesion, we get $SCCM \leq 0.5$. Therefore, I summarize the threshold in table 4.4:

<table>
<thead>
<tr>
<th>Level</th>
<th>SCCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>(0.66,1]</td>
</tr>
<tr>
<td>Acceptable</td>
<td>[0.5,0.66]</td>
</tr>
<tr>
<td>Bad</td>
<td>[0,0.5)</td>
</tr>
</tbody>
</table>
4. Concept

Table 4.4.: SCCM threshold \((0 < N < 6)\)

If \(N \geq 6\), we need to consider the complexity of target component, because high cohesion leads to much complex as well. Therefore, I summarize the threshold in table 4.5.

<table>
<thead>
<tr>
<th>Level</th>
<th>SCCM</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good: Simple module with low complexity and good cohesion</td>
<td>(\frac{2}{N} \leq SCCM \leq \frac{16+2N}{N(N-1)})</td>
<td>When (N \geq 6), (\frac{16+2N}{N(N-1)} &lt; 1), but (\frac{36+2N}{N(N-1)} &gt; 1), it is not allowed because the threshold should be from 0 to 1. After calculating, we get when (N \geq 8), (\frac{36+2N}{N(N-1)} &lt; 1), that means when (N=6) and (N=7), we use 1 to replace (\frac{36+2N}{N(N-1)}).</td>
</tr>
<tr>
<td>Acceptable: Good cohesion, but a little complex</td>
<td>(\frac{16+2N}{N(N-1)} \leq SCCM \leq \frac{36+2N}{N(N-1)})</td>
<td></td>
</tr>
<tr>
<td>Bad: Very low cohesion and low complexity</td>
<td>(SCC \leq \frac{2}{N})</td>
<td></td>
</tr>
<tr>
<td>Bad: High cohesion, but too complex</td>
<td>(SCC \geq \frac{36+2N}{N(N-1)})</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5.: SCCM threshold \((N \geq 6)\)

H.C.M threshold

There is no relative paper giving a similar metric or an evaluating standard for Hybrid Component Cohesion Metric. Therefore, I try to give some assumptions to define the H.C.M threshold.

Assumption 1

Based on the cohesion definition, if a component is cohesive, at least the number of internal relations should equal with the number of coupling relations or be bigger than the number of coupling relations. Because the classes in a component aim to realize a specific function, a outside method invocation mostly leads to a series of internal methods invocation. The internal methods may be in different classes, so a coupling relation causes a series of internal relations. Undeniably, some internal methods may be in a class, that means a coupling relation does not cause a internal relation. Therefore, we assume that one coupling relation leads to one internal relation at least. Then we get \(|Q^R_{\text{Com}_k}| = |O^R_{\text{Com}_k}|\). From the SCCM threshold, we know if \(|Q^R| \leq |P^R|\), the component cohesion is low, so we assume \(|Q^R_{\text{Com}_k}| = |P^R|\). Then we get \(|Q^R_{\text{Com}_k}| = |O^R_{\text{Com}_k}| = |P^R|\), using H.C.M formula to calculate, we get the result H.C.M=1/3=0.33. Therefore, I define H.C.M=0.33 as the minimum threshold.
4.2. Component metrics

Assumption 2

We assume $|Q(R_{Com_k})| = 2|O(R_{Com_k})|$ and $|Q(R_{Com_k})| = |P_R|$, then we get H.C.M = 0.4. If we assume $|Q(R_{Com_k})| = 2|P_R|$ and $|Q(R_{Com_k})| = |O(R_{Com_k})|$, we also get H.C.M = 0.4. While 0.4 is only a little bigger than 0.33, and as mentioned above, $|Q(R_{Com_k})| = |O(R_{Com_k})|$ is the lowest threshold, so 0.4 is not a suitable threshold for describing which result is good.

Assumption 3

We assume $|Q(R_{Com_k})| = 2|O(R_{Com_k})| = 2|P_R|$, actually, it is a good situation and it can represent a highly cohesive component design. Then we get H.C.M = 0.5. Therefore, I define H.C.M = 0.5 as the threshold for a good component cohesion design. I summarize the threshold in the table 4.6:

<table>
<thead>
<tr>
<th>Level</th>
<th>H.C.M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>[0.5, 1)</td>
</tr>
<tr>
<td>Acceptable</td>
<td>[0.33, 0.5)</td>
</tr>
<tr>
<td>Bad (Low cohesion)</td>
<td>[0, 0.33)</td>
</tr>
<tr>
<td>Bad (The Component is useless)</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.6.: H.C.M1 threshold

H.C.M2 threshold

Assumption 1

As described in H.C.M1 Assumption 1, we assume that $|Q(R_{Com_k})| = |O(R_{Com_k})|$ is the acceptable case, then we get H.C.M2 = 0.5, values below 0.5 means low cohesion.

Assumption 2

We assume that $|Q(R_{Com_k})| = 2|O(R_{Com_k})|$ is a good situation, then we get H.C.M = 0.66, values over 0.66 means good cohesion. But when H.C.M2 = 1, that means target component is never used. It is also a bad design. Table 4.7 shows the thresholds about H.C.M2.

<table>
<thead>
<tr>
<th>Level</th>
<th>H.C.M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>[0.66, 1)</td>
</tr>
<tr>
<td>Acceptable</td>
<td>[0.5, 0.66)</td>
</tr>
<tr>
<td>Bad (Low cohesion)</td>
<td>[0, 0.5)</td>
</tr>
<tr>
<td>Bad (The Component is useless)</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.7.: H.C.M2 threshold
4. Concept

CW-H.C.M threshold

Similar with H.C.M, no relative papers discuss the threshold about Call-Weighted Hybrid Component Cohesion Metric. Therefore, I try to give some assumptions to define the threshold.

Assumption 1

As mentioned in H.C.M Assumption 1, basically, a coupling method call need a inner method call in a component at least. But it is not always like this, some coupling calls will not produce any inner calls. So we can assume $|Q^{IR}_k| = |O^{IR}_k|$, this is a situation that can be accepted. Because if a component is designed cohesively, at least the number of inner communication should be bigger than the number of coupling. Then using CW-H.C.W formula, we get CW-H.C.W$=0.5$, It is the minimum threshold. A Call-Weighted H.C.M which is smaller than 0.5 means the cohesion of this component is bad.

Assumption 2

We assume a coupling method call need at least two inner method call in a component, which means that $|Q^{IR}_k| = 2|O^{IR}_k|$, it is a good situation for component cohesion design. Then we get CW-H.C.M$=0.66$, it is another threshold to show that the cohesion of this component is good. I summarize the threshold in table 4.8.

<table>
<thead>
<tr>
<th>Level</th>
<th>CW-H.C.M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>[0.66, 1)</td>
</tr>
<tr>
<td>Acceptable</td>
<td>[0.5, 0.66)</td>
</tr>
<tr>
<td>Bad (Low cohesion)</td>
<td>[0, 0.5)</td>
</tr>
<tr>
<td>Bad (The Component is useless)</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.8.: CW-H.C.M threshold

Threshold integration

Actually, evaluating a component cohesion design based on one metric is not very accurate. That’s why I give three new metrics to measure a component cohesion design from the aspects structure cohesion and function cohesion. I try to combine the three metrics’ results to evaluate a component cohesion design so that we can get a more accurate result.

SCCM and H.C.M

Firstly, I combine the SCCM’ thresholds with H.C.M’ thresholds. Table 4.10 shows the new thresholds when the number of classes in a component is fewer than 6 ($0 < N < 6$). The basic idea of getting the new thresholds is that:
Then we get the new threshold after combining SCCM and H.C.M1.

<table>
<thead>
<tr>
<th>Level</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>$SCCM \geq 0.66 \land 0.5 \leq H.C.M1 &lt; 1$</td>
</tr>
<tr>
<td></td>
<td>$SCCM &lt; 0.66 \land 0.5 &lt; H.C.M1 &lt; 1$,</td>
</tr>
<tr>
<td>Acceptable</td>
<td>$SCCM \geq 0.66 \land H.C.M1 \leq 0.5$,</td>
</tr>
<tr>
<td></td>
<td>$0.5 \leq SCCM &lt; 0.66 \land 0.33 &lt; H.C.M1 &lt; 0.5$,</td>
</tr>
<tr>
<td>Bad (Low cohesion)</td>
<td>$SCCM &lt; 0.66 \land H.C.M1 &lt; 0.33$,</td>
</tr>
<tr>
<td></td>
<td>$SCC &lt; 0.5 \land H.C.M1 &lt; 0.5$</td>
</tr>
<tr>
<td>Bad (Component is useless)</td>
<td>$H.C.M1 = 1$</td>
</tr>
</tbody>
</table>

Table 4.10.: H.C.M1-SCCM threshold ($0 < N < 6$)

I use a x-y plot to show the areas which are split by thresholds, x-axis is H.C.M1, y-axis is SCCM. Figure 4.4 shows the x-y plot when the number of classes is fewer than $6(0 < N < 6)$. The blue area represents the bad design which is low cohesion, the orange area represents the acceptable design, and the green area represents the good design which is high cohesion.
4. Concept

Figure 4.4.: H.C.M1-SCCM PLOT (0 < N < 6)

Table 4.11 shows the new thresholds when the number of classes in a component is equal or more than 6 (N ≥ 6).

<table>
<thead>
<tr>
<th>Level</th>
<th>Threshold</th>
</tr>
</thead>
</table>
| Good                         | \[
\frac{16+2N}{N(N-1)} \leq SCCM \leq \frac{36+2N}{N(N-1)} \land 0.5 \leq H.C.M1 < 1
\] |
| Acceptable                   | \[
\frac{2}{N} \leq SCCM \leq \frac{36+2N}{N(N-1)} \land 0.33 \leq H.C.M1 < 0.5,
\frac{16+2N}{N(N-1)} \leq SCCM < \frac{36+2N}{N(N-1)} \land H.C.M1 < 0.33,
SCCM < \frac{16+2N}{N(N-1)} \land 0.5 \leq H.C.M1 < 1
\] |
| Bad (Low cohesion)           | \[
SCCM < \frac{2}{N} \land H.C.M1 < 0.5,
SCCM < \frac{16+2N}{N(N-1)} \land H.C.M1 < 0.33
\] |
| Bad (High cohesion but high complexity) | \[
SCCM > \frac{36+2N}{N(N-1)}
\] |
| Bad (Component is useless)   | \[
H.C.M1 = 1
\] |

Table 4.11.: H.C.M1-SCCM threshold (N ≥ 6)

Figure 4.11 illustrates the areas which are split by thresholds, the blue area represents the bad design which is low cohesion, the red area represents the bad design which is high cohesion but high complexity. The orange area represents acceptable design, and the green area represents the good design which is high cohesion and not very complex.
The H.C.M2-SCCM threshold is similar with H.C.M1-SCCM threshold. So I give the new threshold directly. Table 4.12 and table 4.13 illustrate the new thresholds when $0 < N < 6$ and $N \geq 6$ respectively. The H.C.M2-SCCM plot is similar with H.C.M1-SCCM plot.

<table>
<thead>
<tr>
<th>Level</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>$SCCM \geq 0.66 \land 0.66 \leq H.C.M2 &lt; 1$</td>
</tr>
<tr>
<td>Acceptable</td>
<td>$SCCM &lt; 0.66 \land 0.66 &lt; H.C.M2 &lt; 1,$</td>
</tr>
<tr>
<td></td>
<td>$SCCM \geq 0.66 \land H.C.M2 \leq 0.66,$</td>
</tr>
<tr>
<td></td>
<td>$0.5 \leq SCCM &lt; 0.66 \land 0.5 &lt; H.C.M2 &lt; 0.66$</td>
</tr>
<tr>
<td>Bad (Low cohesion)</td>
<td>$SCCM &lt; 0.66 \land H.C.M2 &lt; 0.5,$</td>
</tr>
<tr>
<td></td>
<td>$SCC &lt; 0.5 \land H.C.M2 &lt; 0.66$</td>
</tr>
<tr>
<td>Bad (Component is useless)</td>
<td>$H.C.M2 = 1$</td>
</tr>
</tbody>
</table>

Table 4.12.: H.C.M2-SCCM threshold ($0 < N < 6$)
4. Concept

<table>
<thead>
<tr>
<th>Level</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>$\frac{16+2N}{N(N-1)} \leq SCCM \leq \frac{36+2N}{N(N-1)} \land 0.66 \leq H.C.M2 &lt; 1$</td>
</tr>
<tr>
<td>Acceptable</td>
<td>$\frac{2}{N} \leq SCCM \leq \frac{36+2N}{N(N-1)} \land 0.5 &lt; H.C.M2 &lt; 0.66,$</td>
</tr>
<tr>
<td></td>
<td>$SCCM &lt; \frac{16+2N}{N(N-1)} \land 0.66 \leq H.C.M2 &lt; 1$</td>
</tr>
<tr>
<td>Bad (Low cohesion)</td>
<td>$SCCM &lt; \frac{2}{N} \land H.C.M2 &lt; 0.66,$</td>
</tr>
<tr>
<td></td>
<td>$SCCM &lt; \frac{16+2N}{N(N-1)} \land H.C.M2 &lt; 0.5,$</td>
</tr>
<tr>
<td>Bad (High cohesion but high complexity)</td>
<td>$SCCM &gt; \frac{36+2N}{N(N-1)}$</td>
</tr>
<tr>
<td>Bad (Component is useless)</td>
<td>$H.C.M2 = 1$</td>
</tr>
</tbody>
</table>

Table 4.13.: H.C.M2-SCCM threshold ($N \geq 6$)

**H.C.M-SCCM and CW-H.C.M**

Now, I combine the CW-H.C.W thresholds with SCCM-H.C.M thresholds to evaluate the component design. My idea is to use the CW-H.C.M thresholds to evaluate the results getting from SCCM-H.C.M threshold further. For example, if a component design is acceptable based on SCCM-H.C.M, and good based on CW-H.C.W, then I say the component design is good, otherwise, the component design is bad. Table 4.14 shows the final thresholds named Integrity threshold after considering three new metrics.

<table>
<thead>
<tr>
<th>Level</th>
<th>SCCM-H.C.M Level</th>
<th>CW-H.C.W Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excellent</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>Good</td>
<td>Good</td>
<td>Acceptable</td>
</tr>
<tr>
<td></td>
<td>Acceptable</td>
<td>Good</td>
</tr>
<tr>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Acceptable</td>
</tr>
<tr>
<td></td>
<td>Good</td>
<td>Bad</td>
</tr>
<tr>
<td></td>
<td>Bad</td>
<td>Good</td>
</tr>
<tr>
<td>Bad</td>
<td>Acceptable</td>
<td>Bad</td>
</tr>
<tr>
<td></td>
<td>Bad</td>
<td>Acceptable</td>
</tr>
<tr>
<td>Very bad</td>
<td>Bad</td>
<td>Bad</td>
</tr>
</tbody>
</table>

Table 4.14.: Integrity threshold

**4.3. Execution Hot Spot metrics**

In order to solve the question, "What method calls have slowed down the current execution?" and "What classes should be redesigned or recoded in order to optimize
4.3. Execution Hot Spot metrics

execution performance and architecture?", we need to ask more detail questions. Actually, the two questions are about re-factor the software structure. So we need to know why a method or a class should be redesigned. The answer is method calls or classes which delay the current execution or make the structure complex should be redesigned. So a further question is "how the methods calls or classes delay the current execution or make the structure complex?" We also need to consider the data at runtime, so execution frequency and time consuming are two impact factors. If a method call occurred many times and cost much time, this method should be redesigned. If the methods of a class invoked many times by other classes and make the structure complex, this class should be redesigned. Based on these answers, I define that:

- A method which was invoked frequently and it is very time consumed is a method Hot spot.

- A class whose methods were invoked frequently and lead to big delays in a component or between components is a Class Hot Spot.

4.3.1. Identifying the relevant aspects

In the component based software development, the relations between two components and the relations in a component should be considered in two ways on the aspect of software architecture design. If we want to redesign the coupled relations to optimize the software architecture, the methods invoked between components should be considered first. Figure 4.6 shows this situation. Methods getname() and setname() are those methods invoked between component A and component B.

![Figure 4.6.: Situation 1: methods invoked between components](image)

Another situation is shown in figure 4.7. If we want to restructure a component to optimize its functions or simplify the complexity, the methods invoked in this component should be considered first.
4. Concept

4.3.2. Modelling the relevant aspects

The model of Execution Hot Spot Metrics is similar with the model of Component metrics. We have component set $Com_1, Com_2, ..., Component_n \subset Com$, class set $c_1, c_2, ..., c_n \subset C$, method set $m_1, m_2, ..., m_n \subset M$.

4.3.3. Define the scale

Actually, the Execution Hot Spot metrics are based on the method execution frequency, method time consuming. Therefore the scale is from 0 to infinity.

4.3.4. Define the formula

Based on the two situations, I propose two metrics, **Dependency Hot spot Between two components (D-Hot spot)** and **Method Hot spot in a component (M-Hot spot)**. Both of the two hot spot are based on measuring two data, method execution frequency and total time consumed. And the formulas are:

\[
\text{Total execution frequency of a method (TEF}_m) = \sum_{i=1}^{n} EF_i
\]

Where:
- $i$ means a function test or a scenario,
- $n$ means the total number of function tests or scenarios,
- $m$ means the target method name.
- $EF_i$ means the method execution frequency in function test $i$ or scenario $i$.

\[
\text{Class Invoked Frequency (CIF)} = \sum TEF_{m_i}
\]
4.3. Execution Hot Spot metrics

Where:
i means a function test or a scenario,
n means the total number of function tests or scenarios,
m means the target method name.

$TC_i$ means the method time consuming in function test i or scenario i.

After getting the results of TEF and TTC, we can get the D-Hot spot represented by a set including name, TEF, TTC and callee classes, such as (Component B.Class A.method name(), 20, 2000ms, Component A.Class F). And the M-Hot spot is a set including name, TEF, TTC such as (Class A.method name(), 25, 3000 ms). While recoded or restructured methods is not enough to re-factor the software architecture. So I aggregate methods to classes. Figure shows two situations about class hot spot. The left graph shows the class hot spot in a component. I name it as Key Class Hot Spot (KC-Hot Spot). The right graph shows the Boundary Class Hot Spot (BC-Hot Spot), which means the class coupled with other classes in different components. I propose two metrics to support class hot spot. They are:

**Class Invoked Frequency (CIF)**

$\text{Class Invoked Frequency (CIF)} = \sum TEF_{m_i}$

Where:

$m_i$ is a method of target class.

**Class Invoked Time Consuming (CITC)**

$\text{Class Invoked Time Consuming (CITC)} = \sum TTC_{m_i}$

Where:

$m_i$ is a method of target class.

KC-Hot Spot and BC-Hot Spot are both represented as (Class name, CIF, CITC).

In summary, Execution Hot Spot makes it easier for developers and maintainers to understand the performance and architecture problems. Execution Hot Spot is a good way to help optimize software performance and architecture design by identifying method optimization points and component redesign points.
4. Concept

Figure 4.8.: Situations of class hot spot

4.3.5. Define the threshold

The threshold should be flexible, because the metrics are calculated based on execution frequency and time consuming, which are changeable in different software. So the threshold should be defined manually by analysers for different situations.

4.4. Violation metrics

In order to solve the question, "What violations should I address first?", we need to ask some sub-questions. If we want to rank the violations in the behaviour, we need to know:

- What is the difference between violations measured at runtime and that measured in the source code? (Q1)
- What types of violations are there? (Q2)
- What mechanism can be used to rank the violations? (Q3)

4.4.1. Identifying and modelling the relevant aspects

Behaviour VS Static Analysis of violations

For the Q1, I try to find the difference between violations measured in behaviour view and that in static view. While no papers or other materials show that there are some unique types of violations which can only be found in the behaviour view. But after comparing the behaviour view and static view carefully, I think there exists difference between them.

Firstly, some violations can not be discovered in the behaviours view. These violations are caused by inheritance. Like the figure 4.9 shows, class Assistant inherit from class Person. The two classes are in different components. If there is a communication rules which shows there must be no dependency relations between these two components. This inheritance relation is marked as a violation. But as I mentioned in the Motivation part, the inheritance can not be measured in the behaviour view, but can be found through analysing source code.
Secondly, some violations can exist in the source code, but are never executed. That means some violations exist because of some code, but at runtime, these violations are never executed. The reason is the dead code. Sometimes developers will copy some exist code from other classes, while not all the code can be executed in a new class.

Thirdly, the architecture violations have different severity. In the software architecture descriptions, the architects would define the severity for different types of violations. Such as some violations should be forbidden totally, some violations can be acceptable, some violations can be ignored.

Finally, in the behaviour, we can measure how many times a violation take place at runtime. How long a violation delay a software system. But in the static view, these data is impossible to capture.

When I did the evaluation part, I found a special type of violation which should be only measure in the behaviour view. That is caused by Java.lang. reflect. For example, we have two classes. Defining `class Apptest` in source code 4.1 and `class ReflectApp` in source code 4.2. In the static view, there is no direct relation between `class Apptest` and `class ReflectApp`. But at runtime, actually `class ReflectApp` should depend
4. Concept

on class Apptest.

package normal;

public class AppTest {
    private int counter;

    public void printIt(){
        System.out.println("printIt() no param");
    }

    public void printItString(String temp){
        System.out.println("printIt() with param String : " + temp);
    }

    public void printItInt(int temp){
        System.out.println("printIt() with param int : " + temp);
    }

    public void setCounter(int counter){
        this.counter = counter;
        System.out.println("setCounter() set counter to : " + counter);
    }

    public void printCounter(){
        System.out.println("printCounter() : " + this.counter);
    }
}

Source Code 4.1: Apptest.java

package reflection;

import java.lang.reflect.Method;

public class ReflectApp {
    public static void main(String[] args) {
        //no paramater
        Class noparams[] = {};

        //String parameter
        Class[] paramString = new Class[1];
        paramString[0] = String.class;

        //int parameter
        Class[] paramInt = new Class[1];
        paramInt[0] = Integer.TYPE;
        try{

        }
4.4. Violation metrics

```java
//load the AppTest at runtime
Class cls = Class.forName("nomal.AppTest");
Object obj = cls.newInstance();

//call the printIt method
Method method = cls.getDeclaredMethod("printIt", noparams);
method.invoke(obj, null);

//call the printItString method, pass a String param
method = cls.getDeclaredMethod("printItString", paramString);
method.invoke(obj, new String("mkyong"));

//call the printItInt method, pass a int param
method = cls.getDeclaredMethod("printItInt", paramInt);
method.invoke(obj, 123);

//call the setCounter method, pass a int param
method = cls.getDeclaredMethod("setCounter", paramInt);
method.invoke(obj, 999);

//call the printCounter method
method = cls.getDeclaredMethod("printCounter", noparams);
method.invoke(obj, null);

} catch (Exception ex){
    ex.printStackTrace();
}
```

Source Code 4.2: ReflectApp.java

Violation types

For the Q2, now I analyse the violation types based on the general design principles \[SSC10\], back-call principle, skip-call principle and cyclic-call principle. Figure 4.10 shows two types of violations against principles. The red lines represent the Back-Call violations, and the yellow line represents a Skip-Call violation. I didn’t draw a Cyclic-Call violation, because as you see, a Back-Call violation always lead to a Cyclic-Call violation. Generally, we know layer 1 can communicate with layer 2. If the communication layer 2 to layer 1 exists, it is a Back-Call violations, at the same time, it leads to a cycle between layer 1 and layer 2 as well. So it is also a Cyclic-Call violation. Thus, I defines two types of violations based on general design principles, Back-Call violation and Skip-Call violation. And because these principles are well-known and should be followed in the most architecture design, I define them as implicit violation.
In the software architecture descriptions or other documents, architects would give some explicit rules for the architecture communications. For examples, like the figure 4.11 shows, component A can access to component B, but component B can not access to component A, component A in layer 1 can access component C in layer 2, component D in layer 2 can access to component B in layer 1. While we know Component D accessing to component B belongs to Back-Call violations, but in some special cases, the architects would allow it.

From the definition of divergence, we can know that a relation between two components that is not allowed or was not implemented as intended will be checked as a violation. While we need take care of the relations which were not implemented as intended. In implicit rules, they only provide the abstract rules for communication such as which two
4.4. Violation metrics

layers or two components can interact, but have no details such what method can be invoked, which types of parameter can be passed or accessed. So only the explicit rules can be used to check if a relation between two components was implemented as intended. That means a relation may be compliant with implicit rules, but not in explicit rules. Figure 4.12 shows the examples, for implicit rules, we have: Client layer can access to Presentation layer. Like component 1 accesses to component 3 by any method and attribute. For explicit rules, we have: Component 2 can only access to component 4 by method 1. When we check the violations, we find that there is another relation between component 2 and component 4 by method 2. Or there is a relation between component 2 and component 4, but the method are different with the explicit rules.

Another difference between explicit rules and implicit rules is that: Explicit rules will show which components should interact. Like the figure 4.13 shows, the explicit rules define that, component 1 can access to component 2. Component 2 can access to component 4. Component 1 must access to component 4 to get special data directly. But in the implementation, developers did not implement the relation between component 1 and component 4, instead, they use the communication with component 2 to get the data indirectly.

Figure 4.12.: Benefit to differentiate between explicit and/or implicit rules
In summary, I classify architecture violations into two categories. One is implicit violations. It is defined based on the general design principles [SSC10], and it includes two types, Back-call violation and Skip-call violation. Another is Explicit violations, which are defined based on the explicit rules described by architects.

### Weighting for violation types

This part is for Q3. Each type of violations has its own severity. For the explicit violations, architects would give different severity levels for them based on the software requirements. I try to define three severity levels for all explicit violations, high severity, middle severity, and low severity. If there exists some documents to describe the severity level of explicit violations, we can use it directly. If not, architects or developers can have a meet and classify these violations following the three severity levels. For the implicit violations, I try to weight them. A Back-Call violation also leads to a Cycle, so Back-Call violation should have higher severity than Skip-Call violation. For the Back-Call violation, I define two types, Span Back-Call violation and Simple Back-Call violation. Span Back-Call violation means the back call violation taken place in two layers which they are not neighbour. Simple Back-Call violation means the back call violation taken place in two neighbour layers. For the three implicit violations, I define the weights as table 4.15 shows. It is a default ranking proposal, if there exists special cases, we can change it to a new one.

<table>
<thead>
<tr>
<th>Violation type</th>
<th>Weighted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Span Back-Call violation</td>
<td>3</td>
</tr>
<tr>
<td>Simple Back-Call violation</td>
<td>2</td>
</tr>
<tr>
<td>Skip-Call violation</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.15.: Implicit violations’ weights
For the explicit violations, we know if there exists explicit rules, that means they must be followed in this architecture. So those explicit violations should be more severe than implicit violations. So I give the explicit violations weights like table 4.16 shows. Notice that A plus any value equals 0. I make the gap between two levels as 4, because my new metrics will rank the violations based on their weights. I will explain it in the new metrics.

<table>
<thead>
<tr>
<th>Violation type</th>
<th>Weighted</th>
</tr>
</thead>
<tbody>
<tr>
<td>High severity</td>
<td>12</td>
</tr>
<tr>
<td>Middle severity</td>
<td>8</td>
</tr>
<tr>
<td>Low severity</td>
<td>4</td>
</tr>
<tr>
<td>Ignore (allowed)</td>
<td>A (Minus infinity)</td>
</tr>
</tbody>
</table>

Table 4.16.: Explicit violations’ weights

### 4.4.2. Define the scale

Because the violation metrics are based on the times of violation occurred and the weighting values, then the scale should be from 0 to infinity.

### 4.4.3. Defining the formula

After answering the three sub-questions, I propose a new violation metric which is Simple violation weighted value. The metric formula is:

$$\text{Simple violation weighted value} = \text{Explicit violation weighted value} + \text{Implicit violation weighted value}$$

This metric can be used both in static view and behaviour view, it helps developer rank the violations and decide which violation should be solved first. While in the behaviour view, we can measure the method execution frequency, the time cost, so we can consider them as a factor to measure the violations. Thus, I propose another new metrics which can only be used in behaviour view.

$$\text{Execution violation weighted value} = C \times (\text{Explicit violation weighted value} + \text{Implicit violation weighted value})$$

OR

$$\text{Execution violation weighted value} = C \times \text{Simple violation weighted value}$$
4. Concept

The coefficient $C$ can be any factors which can be measured in the behaviour and what the developer want to rank. Such as the number of violations occurred, the delays caused by violations, the maximum delay caused by violations, the minimum delay caused by violations, the average delay caused by violations... If we use the metric **Simple violation weighted value** to measure violations of an architecture, we will get a result list like the figure 4.14 shows. The result value is from 0 to 15. It helps developers know easily and clearly which violation should be addressed first. And for each main area, such as High severity, the different weights represent the violation details. 15 means this violation is an explicit violation and a Span Back-Call violation.

<table>
<thead>
<tr>
<th>Weighted</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>High severity</td>
</tr>
<tr>
<td>..</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Middle severity</td>
</tr>
<tr>
<td>..</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Low severity</td>
</tr>
<tr>
<td>..</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Implicit violations</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Ignore/allowed</td>
</tr>
</tbody>
</table>

Figure 4.14.: Result list of Simple violation weighted value metric

4.4.4. A case study

Now I give an example to show how the two metrics work. Figure 4.15 shows a case of violation measurement. Red arrow represents violations with high severity. Orange arrow represents violations with middle severity. Black arrow represents violations with low severity. Green dotted arrow represents the violations which is implicit violations, but allowed in explicit rules. Black dotted arrow represents violations which are only implicit violations, and not described in explicit rules.
There are five violations, and for each violations, the frequencies of violations occurred are given. Then I use the metric 1 and metric 2 to measure and rank the results like the figure 4.16 shows. Obviously, the ranking results based on metric 1 are different with that based on metric 2. For example, violation I gets the first place in Metric 1 results with 15, but locates in the second place in Metric 2 results. And violation II get the first place in Metric 2 results.

<table>
<thead>
<tr>
<th>Violation</th>
<th>Metric 1 results</th>
<th>Ranking</th>
<th>Metric 2 results</th>
<th>Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>12+3=15</td>
<td>1 --- I</td>
<td>5*(12+3)=75</td>
<td>1 --- I</td>
</tr>
<tr>
<td>II</td>
<td>12+0=12</td>
<td>2 --- II</td>
<td>12*(12+0)=144</td>
<td>2 --- I</td>
</tr>
<tr>
<td>III</td>
<td>8+0=8</td>
<td>3 --- III</td>
<td>6*(8+0)=48</td>
<td>3 --- III</td>
</tr>
<tr>
<td>IV</td>
<td>0+1=1</td>
<td>4 --- IV</td>
<td>5*(0+1)=5</td>
<td>4 --- IV</td>
</tr>
<tr>
<td>V</td>
<td>A+1=0</td>
<td>5 --- V</td>
<td>0</td>
<td>5 --- V</td>
</tr>
</tbody>
</table>

In summary, violations make the software architecture difficult to understand and maintain, also lead to potential problems. Since there are some metrics to measure the violations, but no one ranks them, so I propose two new metrics to rank these violations. The two metrics are basic ideas, they are flexible, the default implicit violation can be adjusted depending on specific development environment.
5. Evaluation

Wherever valour true is found,
true modesty will there abound.

W.S.Gilbert

Contents

5.1. Data collection tools ........................................ 57
5.2. CommunicationIntegrityChecker evaluation ............. 59
5.3. JHotDraw 7 evaluation ....................................... 68
5.4. Summary ...................................................... 72

In order to evaluate the quality of software architecture in behaviour view, a data gathering tool is needed for taking runtime data, I chose CommunicationIntegrityChecker (CIC) which was developed by Johannes Dohmen [Doh13]. CIC was used to extract the runtime data of a software program. After extracting these runtime data, I wrote some small programs in JAVA to filter the data which were needed by these new metric proposals.

Secondly, JhotDraw 7 and CommunicationIntegrityChecker (CIC) were chosen as subject software systems to be evaluated. JHotDraw is an open source software system that shows the implementation of several design patterns, and having been evaluated as having a highly modular decomposition [GP]. JHotDraw 7 is based on JHotDraw, through measuring JHotDraw 7 in a scenario, we cannot only evaluate a part of JHotDraw 7 architecture’s quality, but also validate my proposal metrics by the final results. CIC is a tool for checking the communication integrity of software architecture at runtime. I evaluated its structure and gave some suggestions for improving the architecture quality. I used SCCM, H.C.M and CW-H.C.M to measure the quality of their architectures, and analysed the results to validate the metrics’ thresholds. Execution Hot Spot metrics were also used to analyse the problems of CIC design. Because I have no complete architecture description of JhotDraw 7, so I cannot use Violation metrics to measure the violations. CommunicationIntegrityChecker is a tool to check architecture violations at runtime, the Johannes Dohmen had checked it by the program itself and found no violations. Actually, the Violation metrics are easy to be understood and used, then in the evaluation part, I didn’t validate the Violation metrics.

5.1. Data collection tools

Kieker is a monitoring framework which provides fast and light-weight monitoring and creation of message traces which can be used to extract the required pairs of
5. Evaluation

Execution records. Execution records should include the specific information methods’
execution such as caller method, callee method, start time, end time and so on.
CommunicationIntegrityChecker (CIC) is the main tool for collecting runtime data. It
extracts data from Kieker tracing logs and does further analysis, the output is a JSON
file. The work flow is:

1. Compiling a program with the Kieker instrumentation facilities;

2. Running the program by Kieker monitoring application, getting the Kieker tracing
logs;

3. Using CIC to analyse Kieker tracing logs, getting JSON files.

JSON files (Listing source code 5.1 is a part of a JSON file) provide the data sets that
we need. But JSON files are not the final data sets, because there are many types of
data which are not all useful for metrics input such as 'enhancementState’, 'sequenceID'
and so on. Therefore, these JSON files should be analysed and filtered further. I wrote
some small programs to obtain the necessary data, store them in MySQL database, and
analyse them to compute component metrics. The evaluation work flow is shown in
figure 5.1.

![Figure 5.1.: Evaluation work flow](image)

58
5.2. CommunicationIntegrityChecker evaluation

Since CommunicationIntegrityChecker (CIC) is one of the data collection tools, I evaluate it first. CIC is a program including two applications MonitoringTool and InformationProcessor. The author claims that there were more than one hundred classes (e.g. exception classes or observer interfaces and implementations) in the CIC class library. It is difficult to draw all of them as one big class diagram. So he gave a simplified UML class consisting of the most important packages and their core classes and relations. Figure 5.2 shows the UML class diagram. The core packages are importer, rules, architecture, monitoring, network, validation and exporter.

![Simplified class diagram of the CIC class library (UML)](image)

Then I started to extract the runtime data of CIC. Firstly, using Kieker to run the CIC programs, secondly using CIC to extract the Kieker tracing logs, two JSON files were generated. One is generated by application MonitoringTool, another is by application InformationProcessor. Based on the two JSON files data, I draw a new CIC UML class diagram (figure 5.3) consisting of all packages, their classes and relations which are traced by kieker. Listing source code 5.1 shows one subset data of JSON files. It includes type, caller, callee and so on. And method shows the method’s name, the class name and the package name which the method was belonged to.

```json
1 {  
2   "type": "CALL",  
3   "caller": {  
4     "method": "public void de.rwthaachen.communicationintegrity.marshalling.JsonMarshaller.setEnableNullObjectOutput(boolean)",
5     "start": 1393787678915065804,
6     "end": 1393787678915181429,
7     "enhancementState": "ENHANCEMENT_UNSUCCESSFUL",
8     "enhancement": null  
9   },  
10   "callee": {  
11  
```
5. Evaluation

Source Code 5.1: A data set of JSON files

Obviously, the classes and their relations traced at runtime are different with the static code analysis, table 5.1 compares the number of classes measured at runtime with those measured by static code analysis (Eclipse structure). Some packages include exceptions packages, the classes in exceptions packages aim to deal with exceptions, although they are parts of packages, they are less important than implementation classes, and these classes are used depending whether exceptions occur. Therefore, I did not calculate these exception classes into package classes.

<table>
<thead>
<tr>
<th>Package</th>
<th>#.classes at runtime</th>
<th>#.classes in Eclipse</th>
</tr>
</thead>
<tbody>
<tr>
<td>importer</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>rules</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>architecture</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>marshalling</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>monitoring</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>network</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>validation</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>statistics</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>exporter</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.1.: #.classes at runtime vs real .classes

The reason causing the difference is mentioned in Motivation. The inheritance and interface implementation can not be traced at runtime, in addition, some classes are not used at runtime in this use case. The basic data of CIC was listed in table 5.2. Because CIC had package structure and did not define the components, so I consider a package as a component, the component metrics' results of CIC packages are in table 5.3.
5.2. CommunicationIntegrityChecker evaluation

Figure 5.3.: Class diagram of the CIC drawn based on JSON files
5. Evaluation

| Package   | N | \(|Q^R|\) | \(|P^R|\) | \(|Q'^R|\) | \(|Q''^R|\) | \(|O'^R|\) | \(|O''^R|\) |
|-----------|---|----------|----------|----------|----------|----------|----------|
| importer  | 2 | 0        | 1        | 0        | 0        | 19       | 64       |
| rules     | 7 | 9        | 12       | 10       | 387      | 30       | 1984     |
| architecture | 10 | 13       | 32       | 18       | 952      | 31       | 781      |
| marshalling | 3 | 2        | 1        | 5        | 285      | 26       | 710      |
| monitoring | 8 | 10       | 18       | 17       | 2335     | 44       | 2011     |
| network   | 4 | 3        | 3        | 4        | 371      | 10       | 471      |
| validation | 10| 11       | 34       | 30       | 469      | 41       | 2637     |
| statistics | 3 | 2        | 1        | 7        | 412      | 14       | 611      |

Table 5.2.: Basic data of CIC for component metrics

From table 5.3 we can see every package’s results measured by SCCM, H.C.M1, H.C.M2 and CW-H.C.M. Then I use the Integrity threshold to evaluate these packages’ cohesion.

| Package  | SCCM | H.C.M1-SCCM | H.C.M2-SCCM | CW-H.C.M | Final result
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>marshalling</td>
<td>Good</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Bad</td>
<td>Bad</td>
</tr>
<tr>
<td>network</td>
<td>Acceptable</td>
<td>Bad</td>
<td>Bad</td>
<td>Very bad</td>
<td></td>
</tr>
<tr>
<td>statistics</td>
<td>Good</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Bad</td>
<td>Bad</td>
</tr>
<tr>
<td>rules</td>
<td>Acceptable</td>
<td>Bad</td>
<td>Bad</td>
<td>Poor</td>
<td></td>
</tr>
<tr>
<td>architecture</td>
<td>Acceptable</td>
<td>Bad</td>
<td>Acceptable</td>
<td>Bad</td>
<td></td>
</tr>
<tr>
<td>monitoring</td>
<td>Acceptable</td>
<td>Bad</td>
<td>Acceptable</td>
<td>Bad</td>
<td></td>
</tr>
<tr>
<td>validation</td>
<td>Acceptable</td>
<td>Bad</td>
<td>Bad</td>
<td>Very bad</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3.: Metrics’ results of CIC packages

All the packages’ final results are bad. That means every package’s cohesion of CIC is designed lowly. Actually, SCCM’s value of every package is not bad, that means these

\[ \text{Final result is evaluated based on } 4.14 \]
packages’ structural cohesion are not bad. The problem is the H.C.M and CW-H.C.M. All the packages’ H.C.M are bad. That means every package’s coupling number is much more than its internal relations. Only package architecture and monitoring’s CW-H.C.M are acceptable, others’ are all bad. That means the coupling methods were executed much more frequently than internal methods. So the way to improve the cohesion of CIC packages is to reduce the coupling methods and increase the internal methods. Actually increasing internal methods and reducing coupling methods are interdependent. For example, table [5.5] shows the relations between caller validation and callee rules. Package validation need to invoke methods which were in package rules. Appendix A shows that 7 classes of rules were called by validation and 19 coupling methods of rules are called by validation (See the data in [5.5]). The total number of rules’s coupling relations was 30. Considering the H.C.M formula, we find coupling also has impact on function cohesion. Therefore, reducing the coupling relations between rules and validation is an effective way to make both rules and validation more cohesive in function cohesion. Designing a facade class in a component is an efficient way to increase the internal methods. As well known, facade design pattern is very helpful for information hiding and implementing loose coupling. Further thinking, we find that adding a facade class will increase the number of internal method relations. Because a facade class typically is an interpreter, and aggregates the methods of some internal classes together into itself. Figure [5.4] shows how a facade class work.

Figure 5.4.: Left, components with direct relationships; right, the same components with a facade

For design a facade class, there are two situations. One (Facade 1) is that just design a facade as an interpreter which aggregating methods which were called by other components one by one. Another (Facade 2) is that when design a facade class, merge those methods which were always called together to design new methods. Redesigning some components as one new big component (Merging) is another way to improve the cohesion of a component. If two or several small components were coupled highly, merging them to a new component would be an efficient way to make
the component more cohesive. All the coupling relations among those small components will become the internal relations of the new component.

In Appendix A, we find that `validation->rules` and `marshalling->monitoring` have the most coupling relations. So I propose three solutions for improve their cohesion based on Facade 1, Facade 2 and Merging. Table 5.6 shows the data and final results before and after using proposal solutions. For packages `validation` and `rules`, Facade 1 means design a facad class including all methods invoked by package `validation`. Because `validation` as a caller invoked 19 methods of `rules` which is the callee, and no methods of `validation` were invoked by `rules`. Because there was no change in `validation`, so its result of Facade 1 is the same with Original result, very bad. But `rules` result of Facade 1 becomes Good/Acceptable (because of H.C.M1 and H.C.M2), which is much better than the Original result-very bad. All the values of component metrics become higher. Especially the H.C.M and CW-H.C.M, their values are more than twice as big as Original values. For Facade 2, we need analyse further to locate the problems. From table 5.5, a significant data should be considered, it is 1196 which means class `DroolsValidationTool` called class `DirectedBiCommunicationRule` 1196 times.

<table>
<thead>
<tr>
<th>validation</th>
<th>rules</th>
<th>#.Methods</th>
<th>Total Execution frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DroolsValidationTool</code></td>
<td><code>RuleCatalog</code></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td><code>RuleSet</code></td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td><code>DirectedBiCommunicationRule</code></td>
<td>4</td>
<td>1196</td>
</tr>
<tr>
<td></td>
<td><code>Rule</code></td>
<td>4</td>
<td>571</td>
</tr>
<tr>
<td><code>ValidationResult</code></td>
<td><code>SameSoftwareUnitRule</code></td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td><code>SameArchitectureUnitRule</code></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td><code>Rule</code></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td><code>RuleOperator</code></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td><code>DirectedBiCommunicationRule</code></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>ValidatorTool</code></td>
<td><code>RuleCatalog</code></td>
<td>4</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 5.5: `validation->rules`

Now Execution Hot Spot metrics are needed so that the problem methods can be founded and located. Because the bad design was caused by couplings between `validation` and `rules`. So the Dependency Hot spot Between two components (D-Hot spot) should be measured. In this case, we ignore the time consuming because we aim to reduce the number of couplings. After calculating the Total Execution Frequency of a method \( TEF_m \), we get:

\[
TEF_{getCallee()} = 576
\]

\[
TEF_{getCaller()} = 576
\]

\[
TEF_{getOperator()} = 304
\]

\[
TEF_{getPriority()} = 204
\]
Other methods’ TEF were less than 50, so I did not list the rest methods. Then the D-Hot spots between validation and rules are:

- \((\text{rules.DirectedBiCommunicationRule.getCallee()}, 576, \text{validation.DroolsValidationTool})\)
- \((\text{rules.DirectedBiCommunicationRule.getCaller()}, 576, \text{validation.DroolsValidationTool})\)

If a threshold was defined that methods whose \(\text{TEF} \geq 200\) are D-Hot Spot, then \(\text{getOperator()}\) and \(\text{getPriority()}\) should be added.

- \((\text{rules.Rule.getOperator()}, 304, \text{validation.DroolsValidationTool})\)
- \((\text{rules.Rule.getPriority()}, 204, \text{validation.DroolsValidationTool})\)

Therefore, the four D-Hot spots are parts of factors causing package validation and package rules low cohesion, and should be considered first when re-design validation and rules in order to improve their cohesion. We found an interesting thing, that is the number of execution frequency of \(\text{getCaller()}\) and \(\text{getCallee()}\) are the same. Thus, I read the source code and found that they were invoked together all the time.

Therefore, in Facade 2, we assume that a method \(\text{getCallerCallee()}\) which implements the \(\text{getCaller()}\) and \(\text{getCallee()}\) was designed. Although the result of rules in Facade 2 is also Good/Acceptable, the values of H.C.M and CW-H.C.M is higher than that in Facade 1. The values of validation’s H.C.M and CW-H.C.M also become higher than originally. We have found that Facade 1 and Facade 2 made the rules more cohesive, but affected little on validation. Then I merged rules and validation into one component and measured the new component by metrics. The final result of the new component in Merging is good, which is much better than the Original result—very bad. That means the new component which includes two packages validation and rules is designed as a high cohesion component.

For packages marshalling and monitoring, we assume that a facade class was designed in monitoring, because in \(\text{monitoring->marshalling}\), only one class of marshalling was called by \(\text{monitoring}\). The result of marshalling in Facade 1 is the same with in Original, because there is no change in it. The result of monitoring in Facade 1 is Acceptable, which is better than in Original—Bad. For Facade 2, we use Execution Hot Spot metrics to measure the D-Hot Spot between marshalling and monitoring as well. The D-Hot Spot between marshalling and monitoring are:

- \((\text{monitoring.ExecutionRecord.getTarget()}, 65, \text{marshalling.JsonMarshaller})\)
- \((\text{monitoring.ExecutionRecord.getStart()}, 65, \text{marshalling.JsonMarshaller})\)
- \((\text{monitoring.ExecutionRecord.getEnd()}, 65, \text{marshalling.JsonMarshaller})\)
- \((\text{monitoring.ExecutionRecord.getEnhancementState()}, 65, \text{marshalling.JsonMarshaller})\)
- \((\text{monitoring.ExecutionRecordPair.getType()}, 46, \text{marshalling.JsonMarshaller})\)
- \((\text{monitoring.ExecutionRecordPair.hasCallee()}, 46, \text{marshalling.JsonMarshaller})\)
- \((\text{monitoring.ExecutionRecordPair.hasCaller()}, 46, \text{marshalling.JsonMarshaller})\)
- \((\text{monitoring.ExecutionRecordPair.getSequenceId()}, 46, \text{marshalling.JsonMarshaller})\)
- \((\text{monitoring.ExecutionRecordPair.getSequenceNumber()}, 46, \text{marshalling.JsonMarshaller})\)
- \((\text{monitoring.ExecutionRecordPair.getValidationState()}, 46, \text{marshalling.JsonMarshaller})\)
5. Evaluation

We found that many methods’ execution frequency are the same, then I read the source code and found that methods `getTarget()`, `getStart`, `GetEnd()` and `getEnhancementState()` were called by one class at the same time. Methods `getType()`, `getCaller()`, `hasCallee()`, `hasCaller()`, `getSequenceId`, `getSequenceNumber()` and `getValidationState()` were called by one class at the same time with above methods invoked. In addition, method `getCaller()` which was executed 19 times was also called at the same time. So in the facade class, we assume that two new methods which cover all these methods were designed. The final result of `monitoring` is Good/Acceptable, which is much better than in Original-Bad, and a little better than in Facade 1-Acceptable. Especially the values of H.C.M and CW-H.C.M rise much. In Merging, the final result of the new component consisting of `marshalling` and `monitoring` is good, we can see that each of their final result in Original is bad, but after merging them into a new component, the final result shows that it is a well designed component with high cohesion.

Evaluation by Sonargraph-Architect

Sonargraph-Architect is a powerful tool for analysing software architecture in static view. I use some of its metrics to measure the CIC architecture. Instability (I) is one metric of Robert C.Martin’ metrics, it measures the stability of a component, the value range is from 0 to 1. 0 means very stable and 1 represents very unstable. Results of validation, marshalling were 0.75 and 0.69 respectively, they are very high and represent that validation and marshalling are very unstable. The two packages are not designed as client components, so there are many other packages depending on them, if some changes were done in any of the two packages, many packages would be affected. Therefore, the designs of these two packages is not good. Then I use dependency metrics for further analysis. There were 55 outgoing dependencies from validation to rules, which is the most one in both outgoing and incoming dependencies of validation. There were 12 outgoing dependencies from marshalling to monitoring, which is the most one in both incoming and outgoing dependencies of marshalling. These results shows that validation and rules are coupled highly, and marshalling and monitoring are coupled highly. Obviously, the results measured by static metrics are consistent with that measured by my component metrics in behaviour view.

In summary, every package’s cohesion of CIC is designed poorly. But after redesigning the structure of them by adding facade classes or merging into new components, the final results are improved much. Adding facade classes will improve the cohesion of a component where the facade classes locate, but have little effects on its coupled components. Merging some components into a new component will make the cohesion of this component better, but a problem is that too many classes in a component make the architecture complex and not understandable.
### 5.2. Communication Integrity Checker Evaluation

<table>
<thead>
<tr>
<th>Situation</th>
<th>Original: The original package structure without any changes.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Facade 1: Design a facade class in rules/monitoring.</td>
</tr>
<tr>
<td></td>
<td>Facade 2: Based on Facade 1, optimize the methods in the facade class by aggregating methods which were called at the same time.</td>
</tr>
<tr>
<td></td>
<td>Merging: Combine rules and validation, marshalling and monitoring into new components.</td>
</tr>
</tbody>
</table>

#### Table 5.6: Comparing the results of four situations

<table>
<thead>
<tr>
<th></th>
<th>validation</th>
<th>marshalling</th>
<th>rules</th>
<th>facade 2 rules+</th>
<th>facade 1 rules+</th>
<th>original rules+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>91 37 21</td>
<td>69 16 6</td>
<td>0.70</td>
<td>0.38 0.40 0.22</td>
<td>0.33 0.28 0.20</td>
<td>0.27 0.21 0.15</td>
</tr>
<tr>
<td>Good/Acceptable</td>
<td>96 0 0</td>
<td>90 0 0</td>
<td>0.74</td>
<td>0.48 0.33 0.22</td>
<td>0.35 0.28 0.20</td>
<td>0.27 0.21 0.15</td>
</tr>
<tr>
<td>Acceptable</td>
<td>96 0 0</td>
<td>90 0 0</td>
<td>0.74</td>
<td>0.48 0.33 0.22</td>
<td>0.35 0.28 0.20</td>
<td>0.27 0.21 0.15</td>
</tr>
<tr>
<td>Very bad</td>
<td>96 0 0</td>
<td>90 0 0</td>
<td>0.74</td>
<td>0.48 0.33 0.22</td>
<td>0.35 0.28 0.20</td>
<td>0.27 0.21 0.15</td>
</tr>
<tr>
<td>Very/acceptable</td>
<td>96 0 0</td>
<td>90 0 0</td>
<td>0.74</td>
<td>0.48 0.33 0.22</td>
<td>0.35 0.28 0.20</td>
<td>0.27 0.21 0.15</td>
</tr>
<tr>
<td>Bad</td>
<td>66 0 0</td>
<td>66 0 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Very bad</td>
<td>66 0 0</td>
<td>66 0 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Good/Acceptable</td>
<td>66 0 0</td>
<td>66 0 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Acceptable</td>
<td>66 0 0</td>
<td>66 0 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Very bad</td>
<td>66 0 0</td>
<td>66 0 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Good</td>
<td>19 0</td>
<td>19 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Good/Acceptable</td>
<td>19 0</td>
<td>19 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Acceptable</td>
<td>19 0</td>
<td>19 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Very bad</td>
<td>19 0</td>
<td>19 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Very/acceptable</td>
<td>19 0</td>
<td>19 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Bad</td>
<td>66 0 0</td>
<td>66 0 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Very bad</td>
<td>66 0 0</td>
<td>66 0 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Good/Acceptable</td>
<td>66 0 0</td>
<td>66 0 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Acceptable</td>
<td>66 0 0</td>
<td>66 0 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Very bad</td>
<td>66 0 0</td>
<td>66 0 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Good</td>
<td>19 0</td>
<td>19 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Good/Acceptable</td>
<td>19 0</td>
<td>19 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Acceptable</td>
<td>19 0</td>
<td>19 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Very bad</td>
<td>19 0</td>
<td>19 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
<tr>
<td>Very/acceptable</td>
<td>19 0</td>
<td>19 0</td>
<td>0.33</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
<td>0.13 0.13 0.13</td>
</tr>
</tbody>
</table>

Final Result 2
5.3. JHotDraw 7 evaluation

JHotDraw is an open source software system that shows the implementation of several design patterns, and having a highly modular decomposition. Actually, JHotDraw is a mature JAVA application framework for building graphical drawing editor applications. JHotDraw 7 is based on JHotDraw and defines a JAVA framework for structured drawing editors and for document-oriented applications. The two frameworks can be used independently of each other. There are several sample applications in JHotDraw 7, because JHotDraw 7 is a framework and consists of too many functions, a sample applications is unlikely to cover all classes. Therefore, my strategy is that running a sample application, defining a scenario, and then measuring these packages which were used in this scenario. We only have the package structure of JHotDraw 7 by Eclipse, so these packages are considered as components.

5.3.1. Defining a scenario

JHotDraw Draw is a sample application since the first version of JHotDraw, and it is the basic application of JHotDraw, then I chose JHotDraw Draw as the traced application. The scenario is opening Draw application, drawing two rectangles, inserting a text into every rectangle area, linking two rectangles by a line, finally closing the application. Although it is a simple scenario, its JSON file is 220 MB. It contained 200320 method execution records. The reason is that the Kiker tracing log contained the application initial phase, in addition, it is draw application, so the methods related to mouse listener were invoked much frequently. Figure 5.5 is the package diagram, these packages were used in the scenario. Actually, every package had many sub-packages, in order to make the package diagram clear, only these super packages were drawn.

Figure 5.5.: JHotDraw 7 package diagram
5.3.2. Evaluating by component metrics

The basic data of JHotDraw 7 for component metrics were measured by analysing the JSON file and shown in table 5.7. For package color and undo, only one class was used, then they were ignored in the following measurements.

| Package      | N | |R| | |R| | |R| | |R| |
|--------------|---|---|---|---|---|---|---|---|---|---|---|---|
| sample.draw  | 4 | 3 | 5 | 4 | 114 | 2193 |
| app          | 28 | 59 | 329 | 201 | 536 | 119 | 316 |
| beans        | 2 | 1 | 2 | 4606 | 46 | 1734 |
| color        | 1 | 0 | 0 | 0 | 2 | 169 |
| draw         | 90 | 154 | 3923 | 739 | 44133 | 288 | 10380 |
| geom         | 8 | 7 | 16 | 1245 | 41 | 3007 |
| gui          | 20 | 34 | 158 | 66 | 419 | 46 | 312 |
| undo         | 1 | 0 | 0 | 2 | 0 | 16 | 29 |
| util         | 5 | 1 | 3 | 224 | 178 | 4235 |
| xml          | 2 | 0 | 0 | 0 | 4 | 60 |

Table 5.7.: Basic data of JHotDraw 7 for component metrics

Table 5.8 and table 5.9 show the component metrics’ results of JHotDraw 7 and the final evaluation results respectively. Further analysis of these evaluation results is necessary. Package sample.draw can be considered as a client component, then it needs to communicate with other components frequently, that means it has many couplings, even more than internal relations. It is not a design problem. In addition, the SCCM result of sample.draw was not bad, therefore, sample.draw’s structure is not a bad design. Based on this scenario, actually the main packages that were used should be app, draw and gui. 28 classes were used in package app, SCCM value of app was 0.151, a little bigger than the top threshold 0.122, it means the cohesion of app is high but a little complex. Because the SCCM value is only a little bigger than the top threshold, so the SCCM result can be considered as good. The H.C.M2 value and CW-H.C.M value of app are both 0.63 which are very close to the top threshold 0.66, so the results of H.C.M2 and CW-H.C.M can be considered both good. If so, the evaluation result of app would be excellent, at least app could be considered as a good design. But the H.C.M1 value of app is bad, the reason leading to this case is the number of pair classes which has no direct relations |PR|. In app, |PR| = 329, which affected much on H.C.M1. So when the number of classes in a component is big, especially close to 30, |PR| influences H.C.M1 seriously. 90 classes were used in package draw, while the guideline of architecture smells proposed the limit of the class number in a package was 30. Actually, draw included 15 sub-packages and more than 200 classes, but this scenario only used a part of draw. If we ignore the limit and consider draw as a big component, SCCM value of draw shows it is a highly cohesive component, the results of H.C.M2 and CW-H.C.M are both good, so the final evaluation result of draw is good, which means draw is a good component design with high cohesion but a little complexity. While assume that if we divide draw into several sub-components, every sub-components’ evaluation result would be good. The
5. Evaluation

H.C.M1 value is also very bad, because 90 is a big number and make $|PR| = 3928$ which is a very big number. For package gui, 20 classes were used. Its result of SCCM is 0.177 which means good. The results of H.C.M2 and CW-H.C.M are 0.59 and 0.59 respectively, which are both close to good threshold. Therefore, although the final evaluation result is acceptable, actually gui is very close to good level. While its H.C.M1 is also bad. Its SCCM result is good, but 20 classes make $|PR| = 158$. So after these evaluations, it has proved that H.C.M1 cannot evaluate component design accurately.

<table>
<thead>
<tr>
<th>Package</th>
<th>SCCM</th>
<th>H.C.M1</th>
<th>H.C.M2</th>
<th>CW-H.C.M</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample.draw</td>
<td>0.5</td>
<td>0.041</td>
<td>0.042</td>
<td>0.002</td>
</tr>
<tr>
<td>beans</td>
<td>1.0</td>
<td>0.042</td>
<td>0.042</td>
<td>0.73</td>
</tr>
<tr>
<td>util</td>
<td>0.1</td>
<td>0.016</td>
<td>0.017</td>
<td>0.05</td>
</tr>
<tr>
<td>xml</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.8.: Metrics' results of JHotDraw 7

<table>
<thead>
<tr>
<th>Package</th>
<th>SCCM</th>
<th>H.C.M1-SCCM</th>
<th>H.C.M2-SCCM</th>
<th>CW-H.C.M</th>
<th>Final result(^3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample.draw</td>
<td>Acceptable</td>
<td>Bad</td>
<td>Bad</td>
<td>Bad</td>
<td>Bad</td>
</tr>
<tr>
<td>beans</td>
<td>Good</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Bad</td>
<td>Bad</td>
</tr>
<tr>
<td>util</td>
<td>Bad</td>
<td>Bad</td>
<td>Bad</td>
<td>Bad</td>
<td>Poor</td>
</tr>
<tr>
<td>app</td>
<td>Bad</td>
<td>Bad</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Bad</td>
</tr>
<tr>
<td>draw</td>
<td>Bad</td>
<td>Bad</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Good</td>
</tr>
<tr>
<td>geom</td>
<td>Bad</td>
<td>Bad</td>
<td>Acceptable</td>
<td>Bad</td>
<td>Poor/Bad</td>
</tr>
<tr>
<td>gui</td>
<td>Good</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Acceptable</td>
</tr>
</tbody>
</table>

Table 5.9.: Final results of JHotDraw 7 packages

When I measured JHotDraw 7, some strange direct relations were found, there were all caused by a method named org.jhotdraw.util.prefs.PreferencesUtil.installFramePrefsHandler in package util. For example, at runtime, the execution records show that method installFramePrefsHandler() called the method EventHandler.propertyChange() which located in package draw. In the static view, I used Sonargraph-Architect to measure JHotDrwa 7 and didn’t find such relations. Then I checked the source, but it showed that there is not any direct relation from installFramePrefsHandler() to

\(^3\)Final result is evaluated based on 4.14
5.3. JHotDraw 7 evaluation

`EventHandler.propertyChange()` Firstly, I thought it was caused by listener mechanism of JAVA, but after a test of `actionlistener`, I found that Kieker would not extract the data from JAVA library. Until now, I didn’t find the reason which caused this situation.

Then I used JProfiler to analyse the JHotDraw 7 based on the same scenario which was defined at the beginning, in order to profile the classes which were used in this scenario and get the Hot Spot both in Allocation and CPU. And I compared these results with ones from CIC extracted data. We didn’t monotor Java calls because we have specified a monitoring filter that applies to only the internal JHotDraw 7 packages. But in the JProfiler, there is no such a filter, both Java calls and the JHotDraw 7 internal packages were monitored. JProfiler only provides view filters which are used to filter views that you need, but not filter the call data. In the Memory profiling view, I calculated the packages and their classes based on a sub memory profiling view named "All objects'. Appendix [B] shows the monitored packages and a part of their classes after executing the defined scenario. The packages named like 'org.jhotdraw.app..' are all belonged to package 'org.jhotdraw.app'. Other packages are classified by the same way as well. Some classes like "org.jhotdraw.app.SDIAppli..." and "org.jhotdraw.app.SDIAppli..." is actually a same class. Then after calculating the number of classes of every package, I found the number is smaller but similar with the results measured by CIC. I checked the records extracted by CIC and compared them with ones monitored by JProfiler. Then I found this reason. JProfiler monitored the classes which were instanced at runtime, but CIC traced all classes which were used at runtime, no matter they were instanced or not. For example, in package `geom`, the class `Shapes` was traced, but in the JProfiler, it was not monitored. In CIC extracted data, one of methods in class `Shapes` was called by class `BezierPath`, the detail was that `org.jhotdraw.geom.BezierPath.outlineContains` calls `org.jhotdraw.geom.Shapes.outlineContains`. The source code is in source code 5.2. Through the whole class `BezierPath`, no code showed the class `Shapes` was instanced. But the method `outlineContains()` of class `Shapes` was called directly. Therefore, some of classes were not monitored by JProfiler.

```java
public boolean outlineContains(Point2D.Double p, double tolerance) {
    return Shapes.outlineContains(this, p, tolerance);
}
```

Source Code 5.2: A part of `BezierPath.java`

Appendix [C] shows the results of Allocation Hot Spot which are ranked by allocated memory. Appendix [D] shows the results of CPU Hot Spot which are ranked by inherent time. In the CPU Hot Spot view, the methods’ invocation numbers were given. The method `org.jhotdraw.draw.AttributeKey.hashCode` had the biggest number 17091, I calculated the result of it based on the data extracted by CIC as well, the number was 11156 which was smaller than 17091. The reason is that the invocation number of a method calculated by CPU Hot Spot includes the class internal invocations and Java library classes invocations, while the data extracted by CIC excludes the Java library classes invocations. Because the defined scenario was about drawing some shapes, so we find that no matter the Allocation Hot Spot and CPU Hot Spot, they were all related to the package `draw`. And from the All object view we can see the most monitored classes
5. Evaluation

by JProfiler belonged to package draw, app and gui.

In summary, JHotDraw has been proved as a good framework and having a highly modular decomposition. So the architecture of JHotDraw 7 and every component or package of it are also designed well. Based on the scenario, three main packages app, draw and gui were measured by the three component design metrics. The evaluation results are: app is a good design with high cohesion and a little complexity; draw is a good design, even close to excellent design with high cohesion and a little complexity; gui is close to a good design which good cohesion and simple structure. Therefore, the results measured by the new component design metrics are close to or even the same with experts’ opinions. It proves that the component design metrics are useful, meaningful and at least in the right way.

5.4. Summary

CommunicationIntegrityCheck (CIC) was measured firstly by the component metrics (SCCM, H.C.M, CW-H.C.M), while the evaluation results were all bad. Through further analysis, the problems causing these results were located. Then three solutions were suggested, Facade 1, Facade 2 and Merging. Designing facade classes is a good way to improve the cohesion of a component which is called by others often. Merging components into a new component is an efficient way to redefine a cohesive component. But not all components would be suitable to be combined, because too many classes would lead to complex. So small components and highly coupled components would be suitable to be combined together.

JHotDraw 7 is considered as a good framework with good structure and high modularity. Three packages were measured by component metrics, and finally the evaluation results of these packages were close to good, good or even close to excellent. Therefore, through measuring JHotDraw 7 package structure, the new component metrics are validated as useful and meaningful. In addition, H.C.M1 was validated as an inaccurate metric because of \(|P^R|\) increasing rapidly with the class number rising. Furthermore, two problems of the new component metrics were discovered.

1 These thresholds defined by mathematical way and assumptions are not much accurate, industry experience and experts’ opinions are necessary to make these thresholds accurate.

2 The SCCM value will become small fast with the class number increasing.

Thus, the component metrics and their thresholds should be improved depending on much more experience.
6. Conclusion

If debugging is the process of removing bugs, then programming must be the process of putting them in.

Edsger Dijkstra

6.1. Future Work

These new metric proposals are all designed for measuring the quality of software architecture in behaviour. All the data should be collected at runtime. Component metrics SCCM, H.C.M and CW-H.C.M focus on measuring a component’s cohesion degree at runtime. SCCM measures the component structure cohesion simply. H.C.M and CW-H.C.M measures the component function cohesion. Only one result of the three metrics can not decide if a component is designed with high cohesion, but three results of them can evaluate a component cohesion design comprehensively. In Evaluation part, we have measured the programs CIC and JHotDraw 7 by component metrics. The final results of JHotDraw 7 are consistent with experts’ opinions. Experts evaluated JHotDraw architecture as a good architecture with highly modularity. And by our new component metrics, we evaluated the packages app, draw and gui.

6.1. Future Work

These new metric proposals, Component metrics, Execution Hot Spot metrics and Violation metrics are all in the initial phase. They provide new ways to evaluate the quality of software architecture in behaviour view. While they also need to be improved in the future. One problem which need to be considered firstly is the tool for collecting data which are needed by these metrics. As Johannes Dohmen said, CommunicationIntegrityCheck (CIC) is in its initial phase, and supports only the Kieker framework as a provider of monitored executions. In addition, the output of CIC is a Json file which containing different kinds of data. A data filter program is necessary to be designed for getting the metrics’ input data. It would be a hard job, but it can be realized. Actually, it may be like the Sonargraph-Architect or STAN, the only difference is the data resource, we need runtime data, they collect data from static view. Another big problem is the thresholds of component metrics. The thresholds of SCCM are defined by considering the threshold of McCabe Cyclomatic Complexity metric. While when we use it to evaluate the JHotDraw 7 architecture, we found the top threshold which
defines complexity was not very accurate. Others’ thresholds are mostly defined by assumptions. Actually, these thresholds should defined based on industry experience and experts’ opinions, and for different domain of software architecture, the thresholds may be different. I have an idea for making the thresholds more accurate, if the data collection and filter program was designed and implemented, we can define basic and initial thresholds, the during measuring existed software architectures, the technology machine learning can be used to make the thresholds evolution more accurately.
A. CIC package coupling list
### A. CIC package coupling list

<table>
<thead>
<tr>
<th>Caller-&gt;Callee</th>
<th>Caller (Class name)</th>
<th>Callee (Class name)</th>
<th>#.Execution time</th>
<th>#.methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network-&gt;Monitoring</td>
<td>SendCreatedExecut</td>
<td>ExecutionRecordPa</td>
<td>90</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>recordsThroughXmp</td>
<td>ir</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XmppCommunicator</td>
<td>CreateNewExecutionR</td>
<td>43</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ExecuteRecordsFromXmp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>executionRecordPair</td>
<td>120</td>
<td>3</td>
</tr>
<tr>
<td>Network-&gt;marshalling</td>
<td>XmppCommunicator</td>
<td>JacksonMarshaller</td>
<td>45</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CreateNewExecutionR</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>recordsThroughXmp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network-&gt;validation</td>
<td>SendValidatedExecut</td>
<td>ValidationResult</td>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ionsThroughXmp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>marshalling-&gt;monitoring</td>
<td>JacksonMarshalle</td>
<td>ExecutionRecordPair</td>
<td>342</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>r</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>marshalling-&gt;architecture</td>
<td>JacksonMarshalle</td>
<td>ExecutionRecord</td>
<td>266</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>r</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>marshalling-&gt;architecture</td>
<td>JacksonMarshalle</td>
<td>Unit</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>r</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SoftwareUnit</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RegexSoftwareUnitF</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ilter</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ClassSoftwareUnitF</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>marshalling-&gt;validation</td>
<td>JacksonMarshalle</td>
<td>ValidationResult</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>r</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monitoring-&gt;network</td>
<td>ExecutionRecordFa</td>
<td>SendCreatedExecutio</td>
<td>90</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ctory</td>
<td>nRecordsThroughXmp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monitoring-&gt;validation</td>
<td>ExecutionRecordPa</td>
<td>ValidationState</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ir</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monitoring-&gt;architecture</td>
<td>ExecutionRecordFa</td>
<td>ValidationResult</td>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ctory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monitoring-&gt;marshalling</td>
<td>CreateNewExecutionR</td>
<td>JacksonMarshalle</td>
<td>42</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>ecordsFromXmp</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rules-&gt;architecture</td>
<td>DirectedBiCommunica</td>
<td>Unit</td>
<td>136</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure A.1.: CIC package coupling list 1 of 3
<table>
<thead>
<tr>
<th>Importer</th>
<th>Package</th>
<th>Component</th>
<th>Rule Name</th>
<th>Classes</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>importer</td>
<td>validation</td>
<td>XmlRulesSaxHandler</td>
<td>ValidationResult</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>importer</td>
<td>rules</td>
<td>XmlRulesSaxHandler</td>
<td>RuleCatalog</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>importer</td>
<td></td>
<td></td>
<td>RuleSet</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rule</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SameSoftwareUnitRule</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SameArchitectureUnitRule</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DirectedBiCommunicationRule</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>importer</td>
<td>architecture</td>
<td>XmlRulesImporter</td>
<td>UnitCatalog</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XmlArchitectureImporter</td>
<td>UnitCatalog</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Unit</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>architecture</td>
<td>monitoring</td>
<td>ExecutionRecordEnhancer</td>
<td>ExecutionRecordPair</td>
<td>82</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ExecutionRecord</td>
<td>171</td>
<td>1</td>
</tr>
<tr>
<td>architecture</td>
<td>validation</td>
<td>ExecutionRecordEnhancer</td>
<td>ValidateEnhancedExecutionRecords</td>
<td>41</td>
<td>1</td>
</tr>
<tr>
<td>validation</td>
<td>architecture</td>
<td>DroolsValidationTool</td>
<td>UnitCatalog</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Unit</td>
<td>138</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Validator</td>
<td>SoftwareUnit</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Unit</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>validation</td>
<td>rules</td>
<td>DroolsValidationTool</td>
<td>RuleCatalog</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RuleSet</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DirectedBiCommunicationRule</td>
<td>1196</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rule</td>
<td>571</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure A.2.: CIC package coupling list 2 of 3
A. CIC package coupling list

<table>
<thead>
<tr>
<th>ValidationResult</th>
<th>SameSoftwareUnitRule</th>
<th>12</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SameArchitectureUnitRule</td>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Rule</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RuleOperator</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DirectedBiCommunicationRule</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ValidatorTool</td>
<td></td>
<td>RuleCatalog</td>
<td>40</td>
</tr>
<tr>
<td>validation-&gt;monitoring</td>
<td>Validator</td>
<td>ExecutionRecordPair</td>
<td>163</td>
</tr>
<tr>
<td>ValidateEnhancedExecutionRecords</td>
<td></td>
<td>ExecutionRecordPair</td>
<td>41</td>
</tr>
<tr>
<td>validation-&gt;network</td>
<td>Validator</td>
<td>SendValidatedExecutionRecordsThroughXM</td>
<td>40</td>
</tr>
<tr>
<td>validation-&gt;statistics</td>
<td>Validator</td>
<td>UpdateStatisticsOnRecordValidated</td>
<td>40</td>
</tr>
<tr>
<td>statistics-&gt;monitoring</td>
<td>UpdateStatisticsOnRecordValidated</td>
<td>ExecutionRecordPair</td>
<td>80</td>
</tr>
</tbody>
</table>

Figure A.3.: CIC package coupling list 3 of 3
B. All objects view of JHotDraw 7 by JProfiler
Figure B.1: Packages and classes of JHotDraw 7 monitored by JProfiler
C. Allocation Hot Spot view of JHotDraw 7 by JProfiler
### Figure C.1: Allocation Hot Spot of JHotDraw 7 monitored by JProfiler

<table>
<thead>
<tr>
<th>Method</th>
<th>Allocated Memory</th>
<th>Allocated Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class 1</td>
<td>123 KB</td>
<td>56 KB</td>
</tr>
<tr>
<td>Class 2</td>
<td>345 KB</td>
<td>12 KB</td>
</tr>
<tr>
<td>Class 3</td>
<td>678 KB</td>
<td>23 KB</td>
</tr>
<tr>
<td>Class 4</td>
<td>901 KB</td>
<td>34 KB</td>
</tr>
</tbody>
</table>

The table above shows the allocated memory and the allocated difference for each class. The allocation hot spot is observed in Class 1 with an allocated memory of 123 KB and an allocated difference of 56 KB.
D. CPU Hot Spot view of JHotDraw 7 by JProfiler
Figure D.1: CPU Hot Spot of JHotDraw 7 monitored by JProfiler
E. Database table schema for filter data

In order to collect the necessary data for these new metrics, I wrote some small programs to filter the data from JSON files and store them in MYSQL database. Table E.2 shows a table named basicinformation. It is table for storing the basic information. So there is no primary key or foreign key.

<table>
<thead>
<tr>
<th>Id</th>
<th>Caller</th>
<th>Callee</th>
<th>Starttime</th>
<th>Endtime</th>
<th>relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>varchar</td>
<td>varchar</td>
<td>int</td>
<td>int</td>
<td>varchar</td>
</tr>
</tbody>
</table>

Table E.1.: Basic information table

Table shows a table named relation which only stores the relations from a caller to a callee. The *relation* should be defined as primary key and unique, it cannot be null.

<table>
<thead>
<tr>
<th>Id</th>
<th>relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>varchar</td>
</tr>
</tbody>
</table>

Table E.2.: relation table

If an application will be programmed to analyse the data and calculate these new metric proposals automatically. I suggest to build some views by triggers based on the table basicinformation, such as a view of class table, method table, component table. These views only contain one attribute and the attribute should be designed as primary key and unique. Then these views can be used to support some selection box in the application.
Bibliography


Johannes Dohmen. Validation of architectural communication conformity based on run-time monitoring of software systems. 2013.


Emiliano Tramontana Giuseppe Pappalardo. Suggesting extract class refactoring opportunities by measuring strength of method interactions.


Bibliography


Bibliography


