Towards the Detection and Refactoring of Message Chains in Java Code

***Abstract*— Code smells are patterns in software code that threaten the software's maintainability. In an effort to maintain software quality, these code smells should be detected and refactored. Message Chain is one such code smell which occurs when inefficient responsibility delegations create chains of consequent method calls, resulting in code that becomes difficult to interpret and maintain. This study proposes a method of detecting and suggesting the refactoring for Message Chains. Message Chains are detected through finding method call chains consisting of instances surpassing a threshold number, namely two. Refactoring suggestions are provided by transferring or cloning the target method in the caller Classes. An empirical study is also conducted using Chain Breaker on multiple versions of two popular open-source repositories, Proguard and Incubator-Dubbo-Dubbo. In the study, a correlation between project size (LOC) and the number of Message Chains is found.**

I. Introduction

Software systems need to continuously cope with ever-changing requirements and environments, resulting in constant code modification and the software's evolution. A key requirement for software to evolve is the readability of code, enabling easy interpretation and fast change. But poor or lazy solutions hinder readability. Termed as code smells, these unreadable and unmanageable codes may slow down and threaten the software's evolution, making it difficult for developers to implement necessary changes.

Fowler [1] presented 22 code smells, structures in the source code that may need refactoring. One of those code smells, Message Chains, is analyzed in this study. Message Chains are large chains of method calls caused by improper responsibility delegations in classes. Prior to this work, there have been two existing tools [6][5] for Java, capable of detecting Message Chains. However, neither of these provide automation or suggestions for refactoring.

In this study, a methodology is proposed where Message Chains are detected and refactoring suggestions are provided. Detection is conducted by finding long method call chains and labeling ones with more than two instances as Message Chains. Suggestions for refactoring are developed by creating intermediary methods in the related classes.

An empirical study is conducted to understand the evolutionary properties of Message Chains. Two popular open-source software repositories – Proguard and Incubator-Dubbo-Dubbo – are analyzed to identify a correlation between software project size (measured in KLOC) and Message Chain(s). Three different metrics are used for Message Chains: total Message Chains, Message Chains per KLOC and Maximum Chain Degree. It is found that, as the size of a project increases, so does the number, frequency and degree of Message Chains.

II. Background

Message Chains are created when, in order to complete a task, a method of another class is needed. But in order to reach the needed class, one or more classes are called in between [1]. The Message Chain smell generally arises when a particular class is highly coupled with other classes in chain-like delegations. To illustrate this smell through example, assume that there is Class A which needs data from Class E. Also assume that in order to get the object of Class E, an object of Class D is needed, which can be retrieved from an object of Class C and so forth as described in the sample code snippet below.

$$a.getB().getC().getD().getE().getTheData();$$

Unfortunately in this example, class A is tightly coupled with class B, C and D to reach class E; even though only a reference to Class E is needed. Statements with Message Chain, similar to the one depicted above, can decrease code readability and increase maintenance efforts.

III. Related Work

Fowler et al. [1] identified 22 code smells existing in software systems. They introduced and defined Message Chains as "a long line of *getThis* methods, or as a sequence of temps". Zhang et al. [2] elaborated that definition to include a pattern-based detection process. They mentioned the inclusion of a threshold value which will determine whether a series of "getter" methods or temp variables constitute as a Message Chain.

Zhang et al. [3] empirically analyzed 6 code smells – Duplicated Code, Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man – to find their relationship with faults. They show that Message Chains are associated with a higher numbers of faults in code.

In regards to the detection of code smells, a wide variety of code smell measurement tools have been developed. Tools such as JDeodrant, InFusion, iPlasma and PMD have been developed to detect various code smells such as Feature Envy, God Class and Long Method. However, these tools cannot detect the code smell Message Chain. Fortunately two other tools, Ptidej [5] and the Stench Blossom [6] can successfully detect Message Chains.

Ptidej is a Java application run within the Eclipse IDE. The Ptidej tool suite is integrates various Java packages to provide a comprehensive analysis tool for object-oriented systems. Ptidej creates a system model against which further investigative procedures are run against. Ptidej can be set up to measure the following Fowler smells: Data Class, Large Class, Lazy Class, Long Method, Long Parameter List, Message Chains, Refused Bequest, Speculative Generality.

Stench Blossom is an Eclipse plug-in which produces a graphical indication of the presence of certain code smells and their respective strength. The smells Stench Blossom measures are Data Clumps, Feature Envy, Large Class, Large Method, Message Chain, Switch Statement.

However, neither Ptidej nor Stench Blossom provides automatic recommendation for refactoring smelly code, including Message Chains. This study aims to alleviate this problem by providing developers a way to refactor as well as detect Message Chains.

IV. Proposed Methodology

The methodology consists of two parts – detecting Message Chains and generating suggestions for refactoring those. These are described in the following subsections.

A. Detection Process

As depicted in Algorithm 1, the Message Chain detection process starts by extracting all the executable code from $.java$ files present in the project directory. After the executable code has been loaded, information is extracted from these and saved into custom data structures (Class, Method, Variable and Potential Message Chain Information).

For each method, the executable code inside is checked for potential Message Chains. These are detected using the scope length of method call chains. Scope length (or degree) is calculated using the number of elements in the scope part of a chain. For example, in the chain presented below:

$$a.getB().getC().getD().runTask();$$

$a.getB().getC().getD()$ is the scope part while $runTask()$ is the method call part. With four elements, the scope length for this example is 4. If the scope part for a method call contains two or more elements then the corresponding statement is considered a potential Message Chain.

Potential Message Chains are then filtered based on whether the first element present in the scope corresponds to an instance of classes present in the project directory. Instances of third party library classes are disregarded. After a potential Message Chain is confirmed, its degree is computed and it is added to the list of final Message Chains detected.

As a limitation to this process, private classes are not considered when filtering chains. Only the public Class of the $.java$ files are regarded when processing Message Chains. Furthermore, temp instances are also not considered, and only statements with long chains are included.

B. Refactoring Process

After detecting all the Message Chains, recommendations of refactored code are generated for each of these. The target result of refactoring Message Chains is to reduce the scope part down to a single element. In order to achieve this, new appropriate methods are proposed to be incorporated in classes present along the scope chain. For instance-

Algorithm 1 Pseudo Code for Message Chain Detection

```
1:  for each: method call executable do
2:      if $scope$ exists then
3:          split $scope$ into $elements$;
4:          if $arr[0]$ is project class object and $slope$ length≥2 then
5:              classify as potential Message Chain;
6:  for each: potential Message Chain do
7:      $parentMethod$ = Message Chain container method;
8:      $currentClass$ = parent method's $parentClass$;
9:      split $scope$ into $chainElements$;
10:     for each: $chainElements$ do
11:         if !$lastElement$ then
12:             $nextClass$ = element type;
13:             if !$element$ inner method of $currentClass$ then
14:                 break;
15:         new method name = extract name from remaining element;
16:         parent md's md call = new $md$ name; //md=method
17:         add new $md$ with new $md$ name of next class;
18:         populate new $md$ with remaining $elements$;
19:         append $currentClass$ text to $rs$; //rs = refactor suggestion
20:         if second to $lastElement$ then
21:             append next class text to $rs$;
22:         if $lastElement$ then
23:             extract tail method return type;
24:             propagate tail $md$ return type to new $md$;
```

```
public class A {
    B b;
    public void runA(String a){
        b.getC(a).getD().runTask();
    }
}

public class B {
    C c;
    public void getC(String a){
        return c;
    }
}

public class C {
    D d;
    public void getD(){
        return d;
    }
}

public class D {
    public void runTask(){
        //some task
    }
}
```

Here a Message Chain is present inside the method $runA()$. It needs the method in class D which is accessible through classes C and B. This method chain can be simply alleviated by making the following changes:

```
public class A {
    public B b;
    public void runA(String a){
        b.cDRunTask();
    }
}

public class B {
    public void cDRunTask(String a){
        getC(a).dRunTask();
    }
}

public class C {
    public void dRunTask(){
        getD().runTask();
    }
}
```

Instead of importing ("getting") every instances (B, C and D) needed for accessing the target method ($runTask()$), the importing is delegated to those instances. Every instance will be accessed only by its direct neighbor. To do so, a recursive method is employed:

1) Add a new method ($cDRunTask()$) to the first instance (B) of the chain. The method will contain the original statement starting after the call of the current instance – $getC(a).dRunTask()$.
2) Replace the original call statement ($b.getC(a).getD().runTask()$) with the newly created method – $b.cDRunTask()$.
3) Repeat steps 1 and 2 for all other scope instances except for the last.

The recursion is conducted until the second to last instance of the scope. The last instance (D), which contains the target method ($runTask()$), remains unchanged.

This recursive process adds an intermediary method to the scope instances. Each of these methods contains a call to only its next instance. Hence, no method call statements exceed the degree of 1.

The pre-existing $getter$ methods are not changed or replaced, since these can contain their own business logic. Rather these are reassigned in the intermediary methods. Therefore, the parameters are also unchanged, as seen for $cDRunTask(String\ a)$.

The naming of the intermediary methods is done keeping in mind that meta information may get lost in the process. To avoid that, the class names of the scope instances precede the name of the target method in the new method names.

V. Empirical Analysis

To exemplify the applicability of the process of Message Chain detection and understand its evolution in software system, an empirical study is conducted. Two software projects are empirically analyzed to observe how Message Chains evolve with regards to the size of the projects.

A. Dataset Description

The dataset for the empirical study consists of two open source software with their six major versions. These are mature and easily available software that have adopted the object-oriented paradigm. These two software are-

- **Proguard:** ProGuard[1] is a free Java class file shrinker, optimizer, obfuscator, and preverifier. Versions of this application collected – proguard1.0, proguard2.0.1, proguard3.0.7, proguard4.0, proguard5.0, proguard6.0.
- **Incubator-dubbo-dubbo:** Incubator-dubbo-dubbo[2] is a high-performance, Java based open source RPC (Remote Procedure Call) framework. The versions used are – dubbo2.0.7, dubbo2.0.10, dubbo2.4.10, dubbo2.4.11, dubbo2.5.7, dubbo2.6.4.

Table I shows some metrics of different versions of Proguard and Dubbo. It can be seen that the number of Lines of Code (LOC) has increased significantly with the increase of the versions. Moreover, the number of classes has also increased while the cyclomatic complexity of methods and methods per class have remained similar.

B. Findings

The correlation between project size and Message Chains for the two projects is as follows.

1) **Proguard:** Fig 1 contains the evolutionary properties of Message Chains in Proguard. Fig 1a shows the increasing Message Chain of the Proguard software with the increasing version throughout its lifespan. Fig 1b shows the maximum chain degree also increasing in later versions with the last one having two times its previous version. Fig 1c demonstrates the frequency of Message Chains, showing how Message Chain per $KLOC$ also increase with the software's growth. Fig 1d shows parallel rise of Message Chains and KLOC over the software's lifetime.

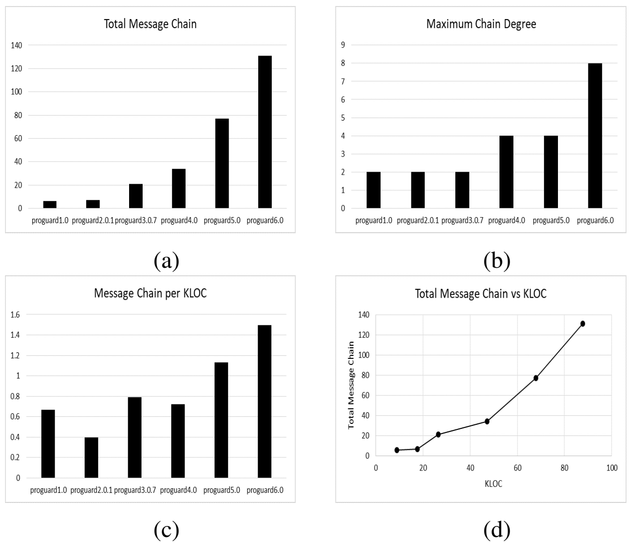


Fig. 1: Evolution of Message Chains in Proguard

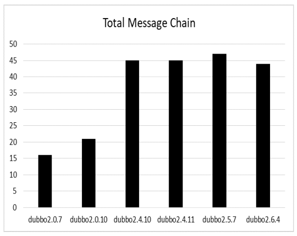
2) **Incubator-dubbo-dubbo:** The evolution of Message Chains in the Dubbo project is provided in Fig 2. Fig 2a shows the increase of the total number of Message Chains. A stark rise can be seen after version

[1]https://sourceforge.net/p/proguard/code/ci/default/tree/
[2]https://github.com/apache/incubator-dubbo/

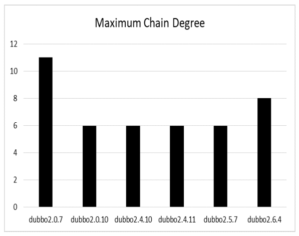
TABLE I: Software metrics of different versions of projects 'Proguard' and 'Dubbo'

Project	Version	LOC	Comment Percentage	Number of Class	Average Cyclomatic Complexiety Per Method	Average Weighted Method Per Class
Proguard	1.0	8995	12.86%	117	1.71	8.62
	2.0.1	17679	13.85%	234	1.69	7.32
	3.0.7	26509	16.01%	294	1.72	15.88
	4.0	47219	17.50%	492	1.82	14.78
	5.0	67914	17.94%	664	1.78	15.14
	6.0	87665	16.92%	796	1.73	9.88
Dubbo	2.0.7	43760	5.19%	589	1.85	7.65
	2.0.10	50614	4.90%	661	1.88	7.59
	2.4.10	89674	5.40%	1091	1.91	7.99
	2.4.11	89678	5.40%	1091	1.91	7.99
	2.5.7	108584	5.41%	1267	2.13	8.41
	2.6.4	97481	4.39%	1295	1.7	7.77

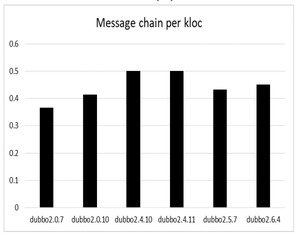
2, but unlike Proguard, the smell remained level with time. In terms of frequency, Fig 2c shows that the project actually improved, reducing Message Chains per KLOC, although not drastically. The last version, as shown in Fig 2d decreased both in size and the number of Message Chains. Lastly, in regards to the maximum degree of Message Chain, it can be seen from Fig 2b) that the initial version had a large chain which was reduced in the following version. In later versions, the degree increased but never too large.



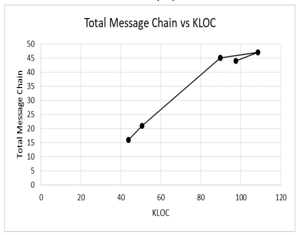
(a)



(b)



(c)



(d)

Fig. 2: Evolution of Message Chains in Dubbo

C. Result Analysis

The number and degree of Message Chains can be helpful in understanding the quality of a developed software. As such, in the empirical study, the projects' life cycles from development inception to the latest stable releases are monitored to understand the impact of change and expansion on software code quality, using the number, frequency and degree of Message Chains as indicators.

The findings show that the number of Message Chains grows with size for both projects, signalling the increase of coupled classes as the system evolve. For Proguard, the increase of Message Chains are quite stark compared to Dubbo, which indicate a lack of refactoring efforts for this particular code smell. The project also shows a steady increase in the maximum size of chains, which can be interpreted as inter-class relations becoming more complex with each consecutive versions.

In the case of Dubbo, Message Chains increase, but at a slower and more stable pace than Proguard. While the total number increases drastically after version 2, the frequency (per KLOC) remains stable, albeit increased. On the other hand, the sharp decline of maximum degree after version 1 indicates a dedicated refactoring effort or redesigning was conducted at that interval. Nevertheless, the maximum degree and presumably the complexity increase with the versions.

VI. Conclusion

In this paper, an approach to detect the Message Chain smell and suggest its refactoring is proposed. An empirical analysis is conducted using this approach to understand the evolution of the smell. It is observed that the amount, frequency and degree of Message Chains increase with the growth of the software system. Future studies can explore varying threshold values and empirically analyze the impact of proposed refactoring method.

References

[1] M. Fowler and K. Beck. Refactoring: improving the design of existing code. Addison-Wesley Professional, 1999.

[2] Zhang, M., Baddoo, N., Wernick, P., and Hall, T. (2008, October). Improving the precision of fowler's definitions of bad smells. In 2008 32nd Annual IEEE Software Engineering Workshop (pp. 161-166). IEEE.

[3] M. Zhang, N. Baddoo, P. Wernick, and T. Hall. Prioritising refactoring using code bad smells. In Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE International Conference on, pages 458 464, 2011.

[4] Moha, N., & Guhneuc, Y. (2010). DECOR: A method for the specification and detection of code and design smells, IEEE Transactions On, 36(1), 2036. https://doi.org/10.1109/TSE.2009.50

[5] Tools Ptidej Team", Ptidej.net, 2018. [Online]. Available: http://www.ptidej.net/tools/. [Accessed: 03- Nov- 2018].

[6] E. Murphy-Hill and A. P. Black. An interactive ambient visualization for code smells. In Proceedings of the 5th international symposium on Software visualization, SOFTVIS 10, pages 514, New York, USA, 2010. ACM.