Profiling call changes via motif mining

Barbara Russo

Free University of Bozen-Bolzano, Italy

barbara.russo@unibz.it

ABSTRACT

Components' interactions in software systems evolve over time increasing in complexity and size. Developers might have hard time to master such complexity during their maintenance activities incrementing the risk to make mistakes. Understanding changes of such interactions helps developer plan their re-factoring activities. In this study, we propose a method to study the occurrence of motifs in call graphs and their role in the evolution of a system. In our settings, motifs are patterns of class calls that can arise for many reasons as, for example, by implementing design choices. By mining motifs of the call graph obtained from each system's release, we were able to profile the evolution of 68 releases of five open source systems and show that 1) systems have common motifs that occur non-randomly and persistently over their releases, 2) motifs can be used to describe the evolution of calls, compare systems and eventually reveal releases that underwent major changes, 3) there are no specific motif types that include design patterns in all systems under study, but each system has motifs that likely include them, motifs that do not include them at all, and motifs that include a design pattern and occur only once in every release. Some of the findings resemble the ones for biological / physical systems and, as such, path the way to study the evolution of call graphs as dynamical systems (i.e., as system regulated by analytic functions).

KEYWORDS

Call Graphs, Motifs mining, Design Patterns, Software Evolution

1 INTRODUCTION

In codebases that consist of hundreds of thousands if not millions of lines of code and complex nets of interactions, the task of adding or changing a piece of code becomes a serious challenge. During maintenance activities, a developer has to know and respect dependencies and interactions with other pieces of code that often someone else wrote. Consequently, there is an increasing risk of making mistakes the more complex a change is to implement in the system [1]. Many tools have been proposed to support developers in their maintenance tasks and mitigate such risk. The majority of such approaches base their algorithms on the syntactic changes of lines of code, which are useful to understand the changes within individual software components like classes (e.g., [2]), but they are less effective to understand the interactions among such components. Recently, graph theory has been used to identify the principles regulating the evolution of components' dependencies [3–5]. In particular, graph theory have been used to model software systems as complex networks by means of the interactions among their components (e.g., as call graphs) [6]. Complex networks are graphs with non-trivial topological features (i.e., different from lattices or randomised graphs whose nodes are randomly connected), which exhibit common global traits: scale-free power law distribution of node degrees and *small-world* properties (i.e., high degree of clustering and short average path length) [7, 8]. Examples of complex networks can be found everywhere: in nature (e.g., biological systems) as well as in engineering systems (e.g., electronic circuits) [6, 7, 9–11]. In social networks, the small-world properties give rise to the popularly-known "six degrees of separation" for which living things in the world are six or fewer steps away from each other. More in general (as for software systems), such common global characteristics facilitate information flow through networks' nodes [3, 6].

Complex networks can be very different in their local organisation instead. In particular, recurring sub-structures can characterise properties of a network or a class of networks that cannot be discovered with the above-mentioned global analysis [7]. Specific recurring sub-structures called *motifs*, have been hypothesised to act as building blocks of software systems or fingerprints of their evolutionary paths [6, 7, 12]. Whether motifs have any functional meaning or a specific role in a system is still an open problem: they may occur as product of intentional design activities (e.g., by implementing a design pattern as Model View Controller), as well as a result of evolutionary rules and maintenance activities (e.g., by duplication-based rule) [13]:

> An interesting question concerns the formation of structures akin to software design patterns. In the same way that recurring spatial patterns (vortices, dislocations, fronts, and solitons) can arise in physical systems under stress, it may be

that recurring functional patterns (adapters, factories, mediators, and proxies) can arise in appropriately defined computational systems driven far from equilibrium. [6]

Worth noticing here that motifs involving three nodes (i.e., size-three motifs) are the first interesting building blocks considered in literature. One of the reasons is that their topology (e.g., closed connect cluster of three nodes) starts being not trivial and their detection and enumeration is not so computationally expensive [14].

Research Goal. The *research goal* of this work is to characterise the evolution of software systems in terms of the types of motifs occurring in the call graphs of all their releases. Such approach is little explored as literature on motifs in software systems is scarce and typically focuses on one single release of a system [6, 13, 15].

Contribution. This work proposes a new method to study the evolution of software systems through the call graphs derived from every system release and use motifs to profile and then compare such graphs. In this way, differences in the evolution of class calls within a system and between systems can be observed. The method is applied to motifs of size three occurring in releases of five open source software systems. Findings show that:

- There are common motifs of size three in the SUTs. Such motifs include the ones found in literature. The new common motifs found in this study, may originate hubs (i.e., classes with large number of dependencies) or cycles (e.g., circular references) in the code. Hubs are classes that are somehow "expected" in software systems, whereas cycles may originate from particular design choices or features of the system.
- Motifs profile and distinguish systems in their evolution. Such profiles may eventually reveal releases that undergo a large change of call interactions.
- Depending on the system, motifs may or may not contain design patterns in all releases. In all systems, there are specific motifs that occur once per release and in all releases and contain an instance of a design pattern.

Structure of the paper. Section 2 introduces the relevant background for our research and provides the major definitions of the paper. Section 3 presents the study design illustrating the goal, research questions and instrumentation for data collection and analysis. Section 4 reports the findings, answering the research questions. Section 5 reviews existing literature relevant for the work, whereas Section 6 discusses the threats that could affect the validity of the results achieved. Section 7 concludes the paper.

2 BACKGROUND

To ease the reading, we introduce here the major concepts we based our work on: call graphs, motifs, and design patterns.

```java
public class Factory{
    int year;
    }
public class DocumentBuilder extends Factory{
    public void buildDocument(){};
    }
public class TextBuilder extends Factory{
    public void buildText(Factory factory,
     DocumentBuilder firstDocumentBuilder){};
    }
public class Catalogue {
    public void createDocuments(DocumentBuilder
     secondDocumentBuilder){};
    }
```
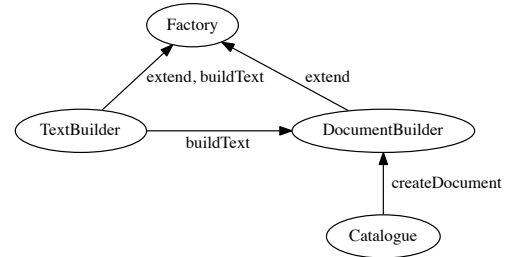

Figure 1: Example of code snippet and its derived call graph.

2.1 Software Systems as Call Graphs

In literature, the dependency structure of software systems has been generally represented in two ways: as **collaboration graph or call graph**. A collaboration graph renders the dependency between classes or objects as data structures (e.g., as in a UML class or object diagram), whereas a call graph represents a run of a program [16]. Call graphs have been used to understand the control and execution flow in software systems, whereas class and/or object collaboration diagrams have been used to gather insight into the relationships among abstract data types in Object Oriented systems. Specifically for call graphs, dependency between classes can further be distinguished between static or dynamic. Static graphs typically describe the set of interactions that are possible, while dynamic graphs identify interactions that actually take place under specific run-time conditions. In this paper, we will focus on **static call graphs**. Different dependency relationships among classes would render the class relationships from another perspective and build different graphs. A call graph is a graph where classes and interfaces are nodes and calls between them are edges, Fig. 1. An edge is directed from the node that calls to the node that is called. To identify calls, we follow the specifications of the Java Virtual Machine [17] as in the following:

- **invokevirtual:** invoking an instance method of an object.
- **invokeinterface:** invoking a method declared within a Java interface. It searches the methods implemented

by the particular runtime object to find the appropriate method,.

- **invokespecial:** invoking an instance method requiring special handling, i.e, an instance initialisation method, a private method, or a superclass method.
- **invokestatic:** invoking a class (static) method.

Listing 1 illustrates an example from JGap 3.6: in its constructor, the class `Chromosome` calls the constructor of its parent class `BaseChromosome` (invokespecial) and invokes statically the method `getStaticConfiguration()` of the class `Genotype` (invokestatic).

Listing 1: JGap 3.6.

```
public Chromosome() throws
    InvalidConfigurationException {
    this(Genotype.getStaticConfiguration());
    }
```

The number of inward (outward) edges of a node is the *in-(out-)degree* or simply degree if no direction for the edge is considered.

Call graphs possess three major properties: (1) scale-free distribution for node degrees, (2) short average path lengths, and (3) high degree of clustering, [13, 18–21]. Properties (2) and (3) define the so-called *small-world networks*. Small-world networks tend to contain highly connected sub-structures (e.g., closed motifs). Property (1) implies two major facts of call graphs: i) the largest node degree is proportional to the size of the network and therefore hubs (i.e., nodes with high degree) likely occur in large graphs, and ii) the degree probability P(k) is power law, $P(k) \sim k^{-\lambda}$ [22]. When quantities follow a power law distribution predicting their global average is not always straightforward. For example, the mean (μ) of a power law distribution is undefined when $\lambda < 2$ and the standard deviation (sd) is infinite for $\lambda < 3$. Thus, if *sd* is unbounded, when we randomly choose a node of the graph, we do not know what to expect: the selected node's degree could be tiny or arbitrarily large [22]. As call graphs of software systems can have λ ranging between 1 and 2.5 [13, 18, 23, 24], the study of the degree distribution in the evolution of call graphs is not that helpful. As an example, Fig. 2 shows the power low distribution we found for util package of the Spring Framework release 3.0.1.

2.2 Motifs

A *network motif* is any interconnected set of nodes of a complex network of a given size and type [7]. The size of a motif is the number of its nodes. In this work, we focus on motifs of size three, i.e., the smallest non-trivial substructures (Fig. 3) as the discovery and enumeration of motifs is computational expensive [14]. In addition, we are interested in design patterns, which typically involve two or three classes and their interactions. The *type* of a motif is defined by its topology, i.e., the direction and the number of edges between nodes. Modulo the number of its directed edges, a motif type is represented by its adjacent matrix, Fig. 3 left. Each entry of the matrix has value 0 or 1 and represents an edge between two nodes. For example, the entry at row = 2 and column = 1
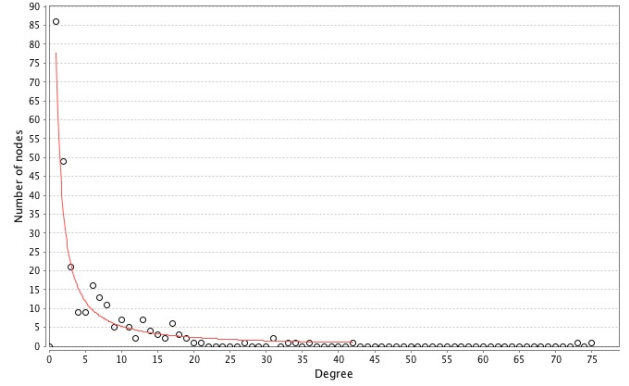
Figure 2: Power law degree distribution for the util package of Spring Framework 3.0.1. Fitted power law line: $y = ax^{-\lambda}$ **with** $a = 77.6$, $\lambda = 1.165$ **and** $R^2 = 0.867$

$$A = \begin{bmatrix} 0(8) & 0(7) & 0(6) \\ \mathbf{1}(5) & 0(4) & 0(3) \\ \mathbf{1}(2) & \mathbf{1}(1) & 0(0) \end{bmatrix}$$
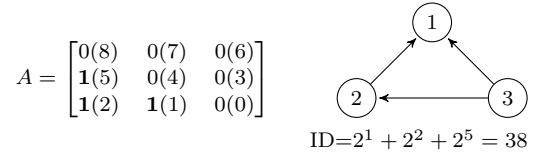
ID$=2^1 + 2^2 + 2^5 = 38$

Figure 3: Adjacent matrix (left), topology type (right) and ID for motif *38*

represents the edge from node 2 to node 1 whereas the entry at row = 3 and column = 2 is the edge from node 3 to node 2. Motifs can have equivalent topologies and therefore the same type. For example, a motif with only two outgoing edges from node 1 is equivalent to a motifs with only two outgoing edge from node 2. The motif ID identifies the type uniquely. The motif ID is computed from the decimal representation of the binary expression of the entries of the adjacency matrix A Fig 3 left. The numbers in brackets in the matrix A represent the bit position. For example, the ID of the motif in Fig 3 right is computed by summing up the bits corresponding to the non zero cells of the adjacent matrix (a_{10}, a_{20}, a_{21}). *In the rest of the paper, we will focus on motifs of size three.* There are 13 different types of size-three motifs. The full list of size-three motifs, their topology, and literature about them is reported in Table 6.

2.2.1 Non-random motifs. To characterise complex network, networks called *random networks* have been used as baseline for comparison [7, 15, 25]. The type of random network we use in this work is the one used in literature of software systems and is built by *randomly re-wiring* the edges of the nodes of the original model (i.e., re-directing links among existing nodes) under the condition that the degree of each node is preserved [8, 26, 27]. Therefore the local connectivity of each node is unchanged as well (e.g., [15, 26, 28]). To characterise an existing network, a Null Hypothesis is first formulated:

NH_0 : *the original complex network and the general random network obtained by link re-wiring have the same number of motifs of size k and type i.*

The effect size to reject the null hypothesis is measured by the *Z-score* (also called *motif significance*). For size k and motif type i, the Z-score is defined by

$$Z_i = (N_i^{real} - <N_i^{rand}>)/std(N_i^{rand})$$

where N_i^{real} is the number of instances of motif of type i and size k in the actual network, and $<N_i^{rand}>$ and $std(N_i^{rand})$ are respectively the mean and the standard deviation of the occurrences of motif of type i and size k in 100 random networks. If Z is greater than two, the null hypothesis can be rejected as the motif type i is significantly overrepresented [7, 10, 15, 29]. One can also use the the P-value to discuss the Null Hypothesis. The P-value represents the probability of a motif to appear an equal or greater number of times in a random network than in the given input network. A motif is usually regarded as statistically significant if the associated P-value is less than 0.01 or z >2. An important difference between the p-value and Z-score is that Z-score can be calculated even if there is a smaller number of random networks. In this article we use the Z-score and two for significance threshold. In this case, the motif is called *non-random motifs*. **In the rest of the paper, we will refer to motifs as non-random motifs if not otherwise stated.**

Finally, Milo *et al.* [30] introduced the so-called *Significance Profile (SP)* to compare networks based on their motifs: for every motif type i

$$SP_i = \frac{Z_i}{(\sum Z_i^2)^{1/2}}$$

and $SP = [SP_1, ..., SP_n]$.

2.3 Design patterns

A design pattern is a sub-structure of a software system that follows a specific design choice and has the goal to detach regulation and control of system functionalities from the objects that finally operate such functionalities. With design patterns, programmers ensure sufficient regulation and control without freezing a system into constraints that are difficult to evolve, [6] and ease evolution [15]. Fig. 4 illustrates the topology of design patterns in terms of classes and their calls as defined in [31].

To identify design patterns in a motif (and a graph), edges that defines template calls in design patterns are "coloured" (i.e., are marked) in the motif (or the graph) [32]. For example, the diagram in Fig 5 shows a motif containing a Proxy pattern: topology (left) and adjacency matrix (right). The proxy method `Request()` corresponds to the dashed edge in the topology and a value greater than one (two in the figure) in the adjacency matrix.

3 STUDY DESIGN

Research Goals and Questions. The *research goal* of this work is to characterise the evolution of software systems in terms of the types of motifs occurring in the call graphs of
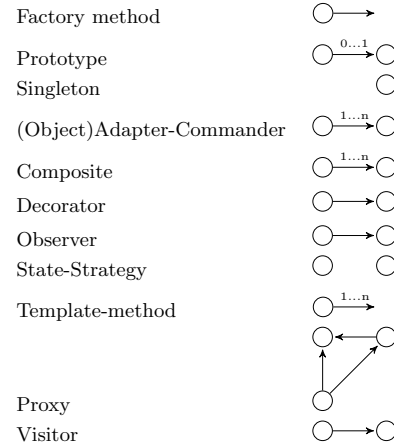

Figure 4: Design Patterns collected by pattern4
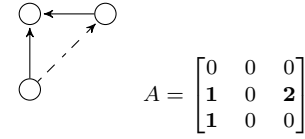

Figure 5: Coloured motif - Proxy pattern

their releases. In particular, we use this characterisation to identify releases whose motifs have been significantly changed. Such analysis is also extended to understand whether the motifs occur by design choices. The results of our work is, therefore, *relevant to* developers that want to identify future releases that need refactoring of class calls or have a better understanding of the effects over releases of design choices or maintenance interventions on the interactions among classes of their systems. For example, such information can be used to understand changes in information flow and, consequently, tune testing strategies [33]. To achieve our goal, we selected subsequent releases of five open source software systems (Spring Framework, JGAP, SquirrelSQL, and Lucene, and JHotDraw) and performed our analysis according to three major questions:

RQ1: *Are there common motifs among releases of the same system under study or among releases of all systems under study?* It has been reported that different software systems share one common motif (motif *38*) [6, 13, 15, 28]. As context, used tools, and settings of the existing studies differ from the one of this work, this question aims at verifying and eventually extending existing findings to call graphs and multiple releases.

RQ2: *Do motifs distinguish systems in their evolution? Do they identify releases that underwent significant changes in the calls among classes?* It has been hypothesised that motifs might reveal stress points of a systems [6]. This question aims at investigating whether motifs can be used to compare
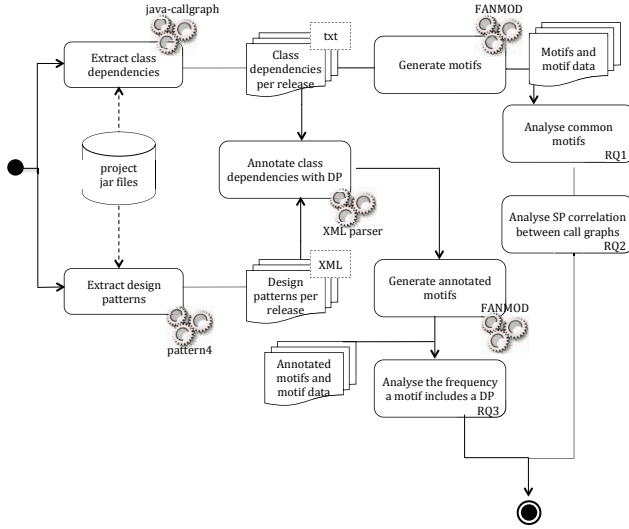
Figure 6: Activity flow of the analysis

systems in their evolution and eventually identify releases that underwent large changes of calls among classes.

RQ3: *To what extent do motifs include design patterns in the systems under study?* According to their templates, design patterns are generally implemented in clusters of 1, 2, or 3 classes Fig. 4. Whether and which of these patterns appear within motifs is still an open question [6]. Motifs may also appear because of developers' intervention during maintenance. These constructs typically emerge to satisfy specific evolution needs (e.g., the addition of a child to a parent class to extend a functionality). We aim at studying motifs to identify the presence of design patterns and emergent sub-structures during maintenance persistently over releases.

3.1 Study method

This section presents a method that extends and integrates previous approaches (Section 5): motifs are detected and studied over releases and systems, design patterns are identified in motifs, and the evolution of systems through the correlations of their SPs between releases is evaluated. Figure 6 illustrates the method and the tools used / developed.

For each release, collect call pairs. We first collect the jar files of each release of the systems from the different repositories of the projects (e.g., github or sourceforge). We identify ordered pairs of classes in a call relationship where the first member calls the second member. To perform this task, we use the tool *javacg-static* [34] that parses the jar file of a release searching for method bodies and returns pairs of classes in a call pairing as described in Section 2.1.

For each release, measure the Z-score of a motif type. For this task, we first assign an ID to each class in a call, convert the pair of class names into pairs of IDs:

$$\texttt{Class.Name}_1, \texttt{Class.Name}_2 \rightarrow [\text{ID}_1] \text{ , } [\text{ID}_2]$$

and then pass the pairs of IDs to the tool for motif discovery *FANMOD* [9, 32]. FANMOD reads IDs pair and output the adjacency matrix and the statistic per motif types. The tool first generates a complex network defined by the call pairs and, by using the RAND-ESU discovery algorithm, it enumerates motifs in such complex network. It then generates 100 random complex networks and enumerates motifs for each of the 100 random networks as well. During enumeration, motifs are also aggregated by their topology type. The final output consists of some basic statistic on the total number of motifs in the original and random networks, the motif frequency, and Z-score per motif type. The RAND-ESU algorithm makes FANMOD faster than similar tool for motif detection [9]. In this study, with a Intel Core i5 with 1.3 GHz CPU and 4Gb of RAM, FANMOD took on average 30s for Spring Framework, 60s for Lucene, 26s for JGAP, 25s for JHotDraw, and 280s for Squirrel for randomisation and motif enumeration per single release.

For each release, identify design patterns in motifs. To collect the instances of design patterns in Fig. 4 for each release of the SUTs, we use the design pattern detection tool, *pattern4* [1] [31]. The output is an XML file containing instances of design patterns per topology type. An instance of Proxy pattern in Gap 3.5 is illustrated in Listing 2.

Listing 2: An instance of a Proxy pattern.

```
<pattern name="Proxy">
    <instance>
        <role name="RealSubject" element="org.jgap.
    FitnessFunction" />
        <role name="Proxy" element="org.jgap.impl.
    BulkFitnessOffsetRemover" />
        <role name="Request()" element="org.jgap.impl.
    BulkFitnessOffsetRemover::clone():java.lang.
    Object" />
    </instance>
</pattern>
```

To recognise instances of design patterns within motif instances as described in Section 2.3, we implemented a Java tool that parses the pattern4 XML output, identifies the template method(s) of the pattern instances, and then annotates the class pairs IDs with a numerical label indicating whether the method is a design pattern method and of which pattern. For example, the instance in Listing 2 is then coded into the triple:

$$[ID_1], [ID_2], [L_1]$$

where

$\texttt{BulkFitnessOffsetRemover} \rightarrow [ID_1]$,
$\texttt{FitnessFunction} \rightarrow [ID_2]$, and
$\texttt{clone()} \rightarrow [L_1]$.

From such triples, FANMOD generates so-called coloured networks and motifs where colours are the labels given to the edges of the network or motif (Section 2.3). In our case, labels correspond to the template methods of design patterns. For example, L_1 labels the method "clone()" as it is the template method "Request()" of the design pattern Proxy in Listing 2. In the generation of coloured motifs, we choose not to colour

---

[1]http://users.encs.concordia.ca/~nikolaos/pattern_detection.html

Table 1: Descriptive analysis of the five systems

System	# rel.	Releases	From - To
SF	37	3.0.1 - 4.1.7	Dec-09 - Jun-15
L	9	2.9.4 - 3.6.0	Dec-10 - Apr-12
S	7	3.0.2 - 3.5.0	June-09 - May-13
JGAP	6	3.4.4 - 3.6.3	Oct-09 - Apr-12
JHD	15	5.2 - 7.6.1	Feb-01 - Jan-11

nodes and set FANMOD to re-wire only edges with the same colour. We choose not to colour the nodes as re-wiring will have little variation in the significance of the motifs: given the size of the motifs the majority of them would not have two nodes of the same colour and it would have been even rarer to re-wire links of the same colour linking nodes with the same colour. Both for coloured and non-coloured networks and for each motif with Z-score greater than two and p-value less than 0.05, and each release, we collected the motif frequency, the total number of motifs in the original network, and the Z-score[2].

3.2 Study Subjects

The analysis is performed on a total of 68 releases of five open source Java software systems that represent different types of software ecosystems: Spring Framework core (SF)[3], Lucene core (L)[4], Squirrel SQL client (S)[5], JGAP (JG)[6], and JHotDraw (JHD)[7]. Table 1 illustrates the systems and their releases. We focused on their major releases, which we identified from the announcement of the corresponding project pages. Releases were also selected to include the ones we already used in our past works (citations omitted for anonymity) so to exploit our previous knowledge of the systems. Figure 7 describes the distributions of nodes, design patterns, classes and edges in the five software SUTs. Worth noticing that there are fewer nodes than classes or interfaces. The difference is due by those classes or interfaces that do not call or are called by other classes. Fig. 8 illustrates the distributions over releases of the most frequent design patterns (Object Adaptor (A), Decorator (D), Proxy (Pr)) in the SUTs (the box plots for all design patterns are omitted for space reason).

4 STUDY RESULTS

RQ1: *Are there common motifs among releases of the same system under study or among releases of all systems under study?* Common motifs of the SUTs are illustrated in Table 2 and Fig. 9. In particular, *motifs 36, 38, 108, 110, and 238 are included in at least one release of any system and all such motifs but motif 238 are common among all releases of all systems,* Fig. 10. In addition, the Z-score plot over releases of motifs *110* or / and *108* dominates all other plots

---

[2]The tools we developed are available at https://goo.gl/gm6NaW.
[3]https://projects.spring.io/spring-framework/
[4]http://lucene.apache.org/
[5]http://squirrel-sql.sourceforge.net/
[6]http://jgap.sourceforge.net/
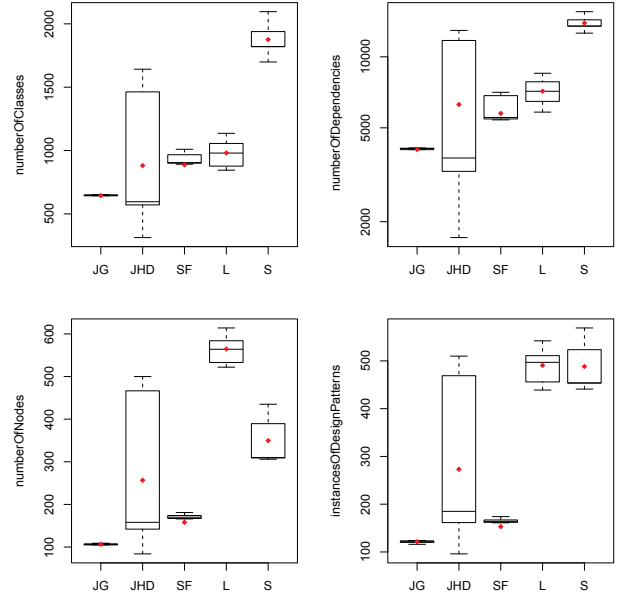[7]http://www.jhotdraw.org/

Figure 7: From top left clockwise: number of classes, links between classes, design pattern instances, and nodes.
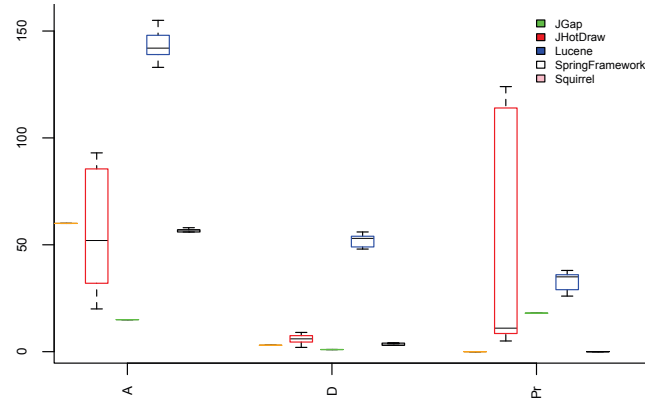

Figure 8: Distribution of instances of the most frequent design pattern in SUTs. Object Adaptor (A), Decorator (D), Proxy (Pr)

Table 2: Common motifs among all releases of SUTs.

Type	JG	JHD	L	SF	S
Common	**36**, **38**, 46, 102, **108**, **110**	**36**, **38**, 46, **108**, **110**	**36**, **38**, 46, **108**, **110**, 238	**36**, **38**, 46, **108**, **110**	**36**, **38**, 98, 102, **108**, **110**, 238
Dominant	108	108	108, 110	108, 110	110

in the SUTs, Fig. 10. Looking at each system, we can see that
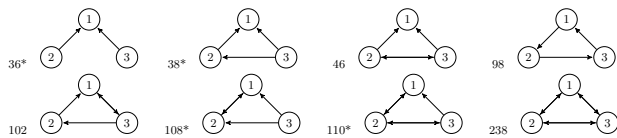
Figure 9: Topology of the motifs appearing in at least one release of one SUT. Motifs labelled with * appears in all releases of all systems.

the plots of all motifs of all releases keep the same order of dominance in JGAP and Squirrel. For JHotDraw, the Z-score of all motifs in release 7.1 drops. According to the API JDiff[8], such drop corresponds to a large removal of packages. Finally, Lucene and Spring Framework exhibit a significant variation of their Z-score plots, Fig. 10.

Comparison with existing literature on motifs of call graphs. *38* is the common size-three motif found in literature of software systems [6, 15, 28]. Such literature has been performed in a context more limited than ours: none of the articles studied motif plots for multiple systems and used the state-of-the-art tool FANMOD [9]. Myers [6] and Ma *et al.* [15] focused on only one release per software system, whereas Petrić and Grbac performed their study on several versions of three Eclipse plug-ins. Myers studied only the frequency of occurrence not the Z-score. Myers and Ma *et al.* used the same motif detection tool mfinder as in [7], whereas Petrić and Grbac compared the average Z-score over releases both with mFinder and Kavosh tool [35]. These authors reported that also motif *46* has a high Z-score for all versions of one plug-in. In our study, such motif was also found in all releases of all systems but Squirrel, Fig. 10. **Motif *38* represents the master regulator-middle managers-workhorse chain of commands defined in [36]**: on top, the the master regulator (zero in-degree) that uses other classes' services, at the bottom, the workhorse (zero out-degree) that provides services to other classes, and in between, the middle managers (non-zero in-degree and out-degree) that mediates services between master regulators and workhorses. This is the typical hierarchy found in the call graph of the Linux kernel for which 58% of the classes are middle managers, 29.6% are master regulators and 12.3% are workhorses [36].

The new common motifs we found reveal some peculiar characteristics of the SUTs. Motif *36* has an open topology (i.e., it is not a triangle) and, as such, it contributes to the occurrence of hub classes with high in-degree (i.e., workhorse hubs, classes providing services to several other classes) [13, 36]. In Section 2.1, we have seen that hubs are expected in large networks indeed. Motif *108*, *110*, and *238* have cycles (i.e., bi-directional edges), which, in general, are not desirable in software systems as they indicate a chain of mutually recursive functions that increase coupling, are hard to understand and test, may eventually turn into stack overflow or deadlocks [37], and may correlate with faults [38]. In some cases, their presence is unavoidable and even necessary,

though. For example, it might have originated from the use of certain design patterns [33]. Thus, removing cycles might sometimes be counterproductive. In our case, a closer look at instances of motif *110* in Squirrel reveals that cycles are often due to nested classes that are kept unchanged over releases. Such motif is also dominant and, thus, very characteristic of the system. Therefore, planning future changes of the classes involved in the such motif can be risky and unnecessary.

There is a set of motifs common to all releases and systems that includes motif *38* found in literature, but not only. Motif *38* represents the typical chain of command of the Linux kernel, whereas the new motifs we found originate workhorse hubs or additional cycles to such chain of command. Workhorse hubs are classes providing services to several other classes, thus removing or modifying them can have a large impact on the system. Cycles can be related to the presence of faults, although, in some systems, they are unavoidable and removing them might be risky.

RQ2: *Do motifs distinguish systems in their evolution? Do they identify releases that underwent significant changes in the calls among classes?* With RQ1, we have discussed common motifs over all systems. With this question, we investigate somehow the opposite: use motifs to distinguish systems in their evolution. To this aim, we analyse the evolution of call graphs through their Significance Profile (SP) (Section 2.2.1). Specifically, we use the correlation between the SP of two releases as a measure of similarity between them. The heatmap in Fig.11 illustrates the effect size of significant correlations ($\alpha = 0.05$) between SPs of all releases of all SUTs. White cells represent non-significant correlations. The darker is the colour the stronger is the effect size. Red squares over the diagonal correspond to correlations of releases of the same system.

Analysis of SP correlations between systems. Releases of JGAP and JHotDraw are similar each other and with the initial six releases of Spring Framework. Lucene's releases are similar to the majority of the releases of Spring Framework and such similarity is stronger after the first six releases. The two systems share the same motifs Fig. 10. After the first six releases, releases of Spring Framework are similar to the one of Squirrel.

Analysis of SP correlations within systems. The map shows the presence of releases that act as transition between set of releases highly correlated. In JGAP, the large number of white cells indicates little similarity of motif profiles of its releases. This might be due to the large variance of the Z-score of motif *238*, Fig. 10. In JHotDraw, there is a clear difference between releases before and after release 7.1 as we already observed above: releases after 7.1 are highly correlated. In Lucene and Squirrel, all releases show high correlation. In Spring Framework, there are four sets within which releases are highly correlated. Such sets are identified by a transition release. Interestingly, the evolution of the SP in Spring Framework has a clear pattern: after the first block, the SPs in the next three blocks are all each other correlated. In other
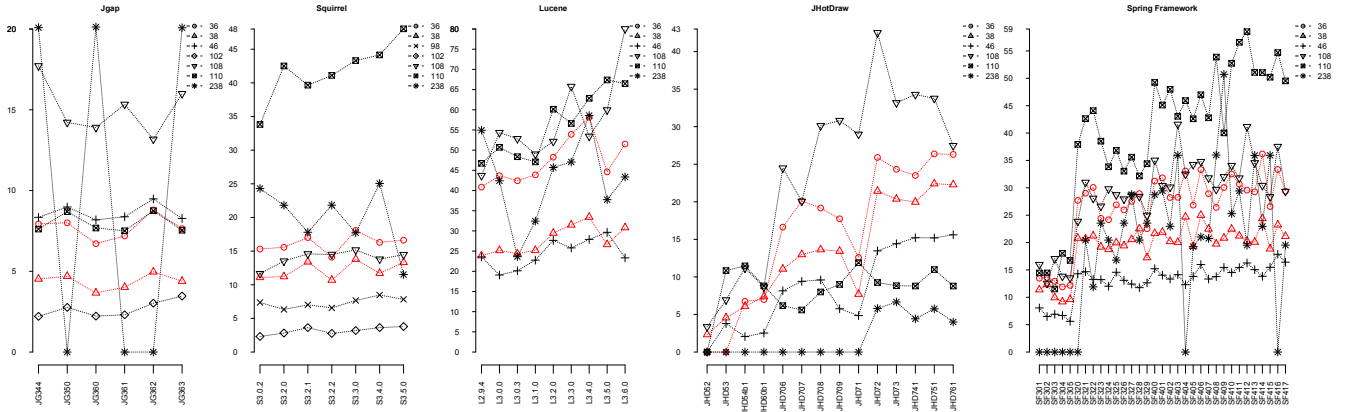
---

[8]https://www.randelshofer.ch/oop/jhotdraw/

Figure 10: Z-score plots of motifs over releases of the SUTs. Red lines indicate the most simple types.

Table 3: % of releases of a system for which $N_{DP}(M) \geq N_{noDP}(M)$ for motif M; "-" indicates no motif instances of type M for all releases of a system.

motif ID	6	12	14	36	38	46	78	102	98	74	108	110	238
JG	-	-	100%	0	0	0	100%	0	-	-	100%	60%	0
JHD	100%	100%	-	7%	0	0	100%	-	-	100%	0	0	0
L	-	-	100%	0	0	0	100%	100%	-	100%	0	0	0
SF	100%	-	-	0	0	0	100%	-	-	-	0	0	0
S	100%	100%	100%	0	0	100%	-	40%	0	100%	0	0	0

words, for the three blocks, after the transition release, Spring Framework restores its structural configuration defined in the second block.
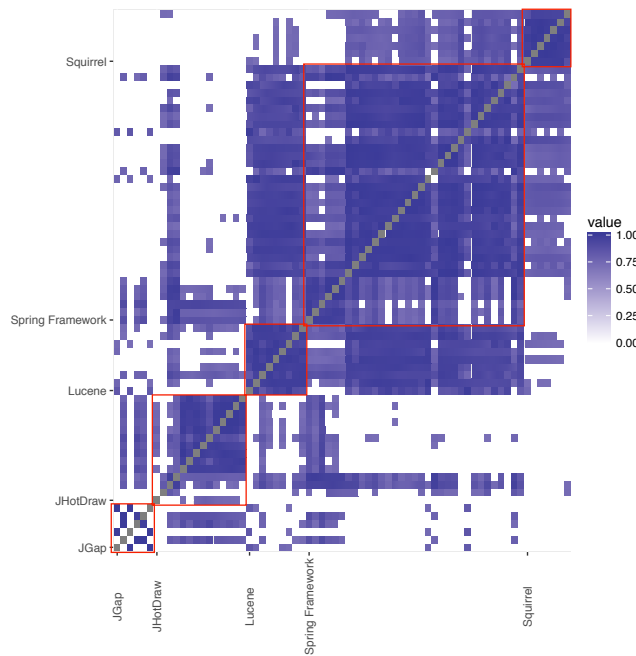

Figure 11: SP correlations between releases.

The **Significance Profile** distinguishes systems in their evolution as only the versions of Lucene show a strong persistent correlation with the mature versions of Spring Framework. The profile may also reveal specific evolutionary patterns: systems whose SP stabilises over versions (JHotDraw Lucene, Squirrel), systems whose SP never stabilises (JGap), and systems that despite large changes of motifs restore to their previous configurations (Spring Framework).

RQ3: *To what extent do motifs include design patterns in the systems under study?* Design patterns determine predefined call interactions among classes and therefore may occur within motif instances. To investigate the occurrence of design patterns in motifs, we start by analysing the percentage of releases for which the number $N_{DP}(M)$ of motif instances of type M that include a design pattern is greater than the number $N_{noDP}(M)$ that do not to include it (i.e., such releases likelier include design patterns). Table 3 illustrates the percentage of releases x for which $N_{DP}(M) \geq N_{noDP}(M)$ in the five SUTs. According to the Table, motifs *6, 12, 14, 74, 78* have x=100% for two or more systems. A closer look at their count (omitted) tells that these motifs typically occur in one (the same) instance in each release and some of them include cycles. Instances of motifs *46, 102, 108, 110* are more frequent (count omitted) and likely contain design patterns in all releases of one system, Table 3. For example, motifs *108* and *110* likelier contain design patterns in JGAP and motifs *46* and *102* in Squirrel. On the other hand, motifs *38, 238* less likely contain design patterns (as $N_{DP}(M) < N_{noDP}(M)$)

for all releases of all five systems. Table 4 illustrates the percentage of releases for which instances of a given motif do not contain design patterns at all. As expected, motifs *6, 12, 14, 74, 78* do not appear in the Table. Instances of motif *36* do not include design patterns in some releases of all systems (e.g., in all releases of Squirrel and in 88.9% of the releases of Lucene). Such percentage varies depending on the system. For Squirrel, for example, motif *36* and *98* do not include design pattern instances in all its releases.

Table 4: % of releases that have no design patterns in all motif instances of a given type

motif ID	JG	JHD	L	SF	S
36	50%	26.70%	88.90%	12.50%	100%
46		26.70%			
98					100%
102	66.60%				
108		60%		93.80%	80%
110		33.30%		6.30%	
238	33.30%	33.33%		75%	

Overview. Table 5 summarise the major results obtained by the analysis of RQ1 and RQ3. Motif *98,108, 238* do not include a design pattern for a large number of releases. Worth noticing that **dominant motifs may or may not contain design patterns over releases.** For example, motif *108* likely contains design patterns for JGAP, but does not contain design patterns for JHotDraw, Spring Framework, and Squirrel. In table 5, we can see that **design patterns are always contained in motifs *6, 12, 74, 78*, which in turn occur in every release and once per release.** This is particularly the case for JHotDraw that, more than the other systems, embraces the use of design patterns in the development. Thus, it appears that *such motifs are reserved for the single design patterns and have no topological variation during the evolution of a system.* Although we cannot prove that no variation of such motifs is due to the presence of the single design pattern instance, it is worth noticing that recent findings in [39] report that classes participating in single design patterns tend not to change over time due to the design dependencies with other classes. Thus, some design choices seem to constraint the evolution and variability of a system.

5 RELATED WORK

Evolution of call graphs. The evolution of call graphs has been long studied in terms of the power law of the node degree distribution: Myers [6] studied both collaboration graphs and call graphs; Vasa et al. [24, 40] built a tool to detect static interactions among classes and analysed call graphs of 12 Java projects; Wang *et al.* [23] focused on the call graphs of 223 versions the Linux kernel; Bhattacharya *et al.* [37] analysed 11 large projects over multiple releases; in 2015, Wang and Ding [18] studied the Linux kernel. Worth noticing here that the literature on power law for node degree distribution has also concerned collaboration graphs (e.g., Valverde and Solé

2005 [13], de Moura et al. [41]), but such articles are out of the scope of this study.

Motifs and motifs' evolution in call graphs. According to our settings, we limit our literature review to research on size-three motifs in call graphs. To our knowledge, such literature dates back to 2003 [6] when Christopher Myers conjectured that motifs could be used to understand evolutionary rules of software systems. He also briefly mentioned that call graphs have the common size-three motif *38*. Later, Petrić and Grbac [28] studied the evolution of size-three motifs over several releases (of three Eclipse plug-ins) and found that motif *38* occurs in all the releases of the plug-ins, whereas motif *46* in all releases of one plug-in. They also showed that the two algorithms (Kavosh [35] and mFinder [42]) detect a different number of motifs with Kavosh outperforming mFinder. They also built call graphs using an in-house made tool that collects slightly different parameters (i.e., method invocation, return type or parameters) from the javacg-static tool. Ma *et al.* [15] studied motifs of one release of six systems and found that motif *38* was the most common. They collected motifs with mFinder and filtered out motifs that seldom occur.

Design patterns and motifs of call graphs. There is no literature on motifs and design patterns for call graphs. Recently, Ampatzoglou *et al.* [39] extensively studied the effect of design patterns on changes of collaboration dependencies. They found that depending on their role, classes in design patterns tend to resist to change.

Other studies on call graph analysis. Zimmermann and Nagappan [38] analysed the complexity of the interactions among and within call graphs. Measures of package internal / external interactions were defined and correlated to number of failures. Steff and Russo modelled systems as call graphs and introduced a measure of similarity based on bit representation of class neighbourhoods [4]. Yan *et al.* [36] compared the call graph of a version of the Linux kernel with transcriptional regulatory network of genomes in terms of their hierarchical layout.

6 THREATS TO VALIDITY

Given the explorative and observational nature of our work, the relevant class of threats is *construct validity*. The major thread of any work on motif detection is the limitation of the algorithm. To mitigate this threat we used the state-of-the-art tool, FANMOD [9, 32]. A second important threat is the random network used in the definition of the Z-score. Van Nes *et al.* [26] argued that a better null model takes into account the dynamic characteristics of a network. For metabolic systems, they constructed such model by better defining the strength of the relations between two nodes. This may hold true also in the case of software systems although the results with random re-wiring are promising.

Finally, as for any study performed on a finite number of systems, we cannot claim any *external validity* as our results may not hold true for other systems although our work is in line with existing literature.

Table 5: Motif analysis' overview. In brackets, the percentage of releases.

Motif's characteristic	JG	JHD	L	SF	S
Dominant	108	108	108, 110	110	110
Likely contain DP	108(100%), 110(60%)	36(7%)	102(100%)		102(40%), 46(100%)
No DP	36(50%), 102(66.6%), 238(33.3%)	36(26.7%), 46(26.7%), 108(60%), 110(33.3%), 238(33.3%)	36(88.9%)	36(12.5%), 108(93.8%), 110(6.3%), 238(75%)	36(100%), 98(100%), 108(80%)
Single DP instance	14, 78	6, 12, 74, 78	14, 74, 78	6, 78	6, 12, 14, 74

7 CONCLUSIONS AND FUTURE WORK

In this work, we have proposed a method to study the evolution of motifs over releases and characterise them as known design patterns or emergent sub-structures. We further used motifs to distinguish releases that underwent large changes in class calls and compare systems during their evolution. We applied our method to five open source software systems. Given the explorative nature of our work and the computational limits of the enumeration algorithms, we focused our analysis on size-three motifs. We found that the systems have motifs that occur non-randomly and persistently over their releases, the set of motifs can be used to describe the overall evolution of call changes and compare systems, and there are no specific motif types that include design patterns in all SUTs, but each system has motifs that likely include them, motifs that do not include them at all, and motifs that occur only once in every release and include a design pattern. At this stage of the work, we cannot associate to any motif type a unique role in the logic of a system. For example, motif of type *36* can originate from inheritance relations as well as simple associations between classes. One possible way to gain better understanding on this issue might be to infer from motifs new design patterns following the approach in [43]. Such approach will be matter of future work.
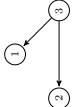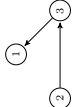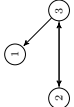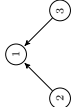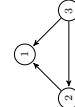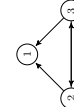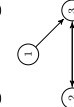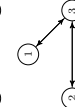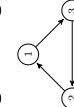
The replication of our work to size-4 motifs or higher will be the first straightforward extension of this work, although it will req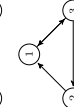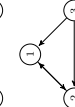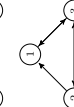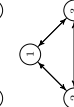uire much more computational power or parallelisation. Another extension of our work concerns the re-definition of the null model for the Z-score. The re-wiring mechanism may produce a too basic null model not representing the real nature of call graphs. Non-trivial null models have been already introduced for biological systems, [26], and we plan to replicate this approach for software systems. Such open issue is also related to a more ambitious goal of our future research: understanding the dynamic ruling the evolution of call graphs both as Self-Organised Criticality process [3] or dynamic systems [26].

REFERENCES

[1] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070510

[2] B. Rossi, B. Russo, and G. Succi, "Analysis of open source software development iterations by means of burst detection techniques," in *Open Source Ecosystems: Diverse Communities Interacting, 5th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2009, Skövde, Sweden, June 3-6, 2009. Proceedings*, 2009, pp. 83–93. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02032-2_9

[3] Z. Lin and J. Whitehead, "Why power laws? an explanation from fine-grained code changes," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 68–75.

[4] M. Steff and B. Russo, "Measuring architectural change for defect estimation and localization," in *Empirical Software Engineering and Measurement (ESEM), 2011 ACM-IEEE International Symposium on*, 2011, pp. 225–234.

[5] L. Moonen, S. Di Alesio, T. Rolfsnes, and D. W. Binkley, "Exploring the effects of history length and age on mining software change impact," in *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*, 2016, pp. 207–216. [Online]. Available: http://dx.doi.org/10.1109/SCAM.2016.9

[6] C. R. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs," *Phys. Rev. E*, vol. 68, Oct 2003.

[7] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: simple building blocks of complex networks," *Science's STKE*, vol. 298, no. 5594, p. 824, 2002.

[8] D. J. Watts and S. H. Strogatz, "Collective dynamics of "small-world" networks," *Nature*, vol. 393, pp. 440–442, 1998.

[9] N. Tran, S. Mohan, Z. Xu, and C. Huang, "Current innovations and future challenges of network motif detection," *Brief Bioinform*, vol. 16(3), pp. 497–525, 2015.

[10] A. L. Robert J Prill, Pablo A Iglesias, "Dynamic properties of network motifs contribute to biological network organization," *PLoS Biol*, vol. 3, no. e343, 2005.

[11] S. Maslov and K. Sneppen, "Specificity and stability in topology of protein networks," *Science*, vol. 296, 2002. [Online]. Available: http://dx.doi.org/10.1126/science.1065103

[12] S. Valverde and R. V. Solé, *Motifs in Graphs*. New York, NY: Springer New York, 2009, pp. 1–15. [Online]. Available: https://doi.org/10.1007/978-3-642-27737-5_339-3

[13] S. Valverde and R. V. Solé, "Network motifs in computational graphs: A case study in software architecture," *Phys. Rev. E*, vol. 72, Aug 2005.

[14] S. Wernicke, "Efficient detection of network motifs," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 3, no. 4, pp. 347–359, Oct. 2006.

[15] Y. Ma, K. He, and J. Liu, "Network motifs in object-oriented software systems," *Dynamics of Continuous, Discrete and Impulsive Systems (Series B: Applications and Algorithms)*, vol. 14(S6), pp. 166–172, 2007.

[16] B. G. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 216–226, May 1979.

[17] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, May 3, 2014 - Computers, 2014.

[18] Y. F. Wang and D. W. Ding, "Topology characters of the linux call graph," in *2015 2nd International Conference on Information Science and Control Engineering*, April 2015, pp. 517–518.

[19] M. E. J. Newman, "Power laws, Pareto distributions and Zipf's law," *Contemporary Physics*, vol. 46, pp. 323–351, Sep. 2005.

[20] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 1, pp. 2:1–2:26, Oct. 2008. [Online]. Available: http://doi.acm.org/10.1145/1391984.1391986

[21] S. Valverde and R. Solé, "Hierarchical small worlds in software architecture," *Dynamics of Continuous, Discrete and Impulsive Systems Series B, Special Issue on Software Engineering and Complex Networks*, vol. Supplement, Vol.14(S6), 2007. [Online]. Available: http://arxiv.org/abs/cond-mat/0307278v2

[22] A. Barabási and M. PÃsfai, *Network Science*. Cambridge University Press, 2016.

[23] L. Wang, Z. Wang, C. Yang, L. Zhang, and Q. Ye, "Linux kernels as complex networks: A novel method to study evolution," in *2009 IEEE International Conference on Software Maintenance*, Sept 2009, pp. 41–50.

[24] R. Vasa, J. G. Schneider, C. Woodward, and A. Cain, "Detecting structural changes in object oriented software systems," in *2005 International Symposium on Empirical Software Engineering, 2005.*, Nov 2005, pp. 8 pp.–.

[25] J. F. Knabe, C. L. Nehaniv, and M. J. Schilstra, "Do motifs reflect evolved function?—no convergent evolution of genetic regulatory network subgraph topologies," *Biosystems*, vol. 94, no. 1–2, pp. 68 – 74, 2008.

[26] P. van Nes, D. Bellomo, M. J. T. Reinders, and D. de Ridder, "Stability from structure: Metabolic networks are unlike other biological networks," *EURASIP J. Bioinformatics Syst. Biol.*, vol. 2009, pp. 4:1–4:15, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1155/2009/630695

[27] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, no. 2, p. 026113, Feb. 2004. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.69.026113

[28] J. Petrić and T. G. Grbac, "Software structure evolution and relation to system defectiveness," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE2014, 2014.

[29] Y. Artzy-Randrup, S. J. Fleishman, N. Ben-Tal, and L. Stone, "Comment on "network motifs: Simple building blocks of complex networks" and "superfamilies of evolved and designed networks","" *Science*, vol. 305, no. 5687, pp. 1107–1107, 2004. [Online]. Available: http://science.sciencemag.org/content/305/5687/1107.3

[30] R. Milo, S. Itzkovitz, N. Kashtan, R. Levitt, S. Shen-Orr, I. Ayzenshtat, M. Sheffer, and U. Alon, "Superfamilies of evolved and designed networks," *Science*, vol. 303, pp. 1538–1542, 2004.

[31] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design pattern detection using similarity scoring," *Software Engineering, IEEE Transactions on*, vol. 32, no. 11, pp. 896–909, 2006.

[32] S. Wernicke and F. Rasche, "Fanmod: a tool for fast network motif detection," *Bioinformatics*, vol. 22, no. 9, pp. 1152–1153, 2006.

[33] T. D. Oyetoyan, J. R. Falleri, J. Dietrich, and K. Jezek, "Circular dependencies and change-proneness: An empirical study," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 241–250.

[34] G. Gousios. java-callgraph: Java call graph utilities. [Online]. Available: https://github.com/gousiosg/java-callgraph

[35] Z. R. M. Kashani, H. Ahrabian, E. Elahi, A. Nowzari-Dalini, E. S. Ansari, S. Asadi, S. Mohammadi, F. Schreiber, and A. Masoudi-Nejad, "Kavosh: a new algorithm for finding network motifs," *BMC Bioinformatics*, vol. 10, no. 1, p. 318, 2009.

[36] K.-K. Yan, G. Fang, N. Bhardwaj, R. P. Alexander, and M. Gerstein, "Comparing genomes to computer operating systems in terms of the topology and evolution of their regulatory control networks," *Proceedings of the National Academy of Sciences*, vol. 107, no. 20, pp. 9186–9191, 2010.

[37] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 419–429.

[38] T. Zimmermann and N. Nagappan, "Predicting subsystem failures using dependency graph complexities," in *Proceedings of the The 18th IEEE International Symposium on Software Reliability*, ser. ISSRE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 227–236. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2007.19

[39] A. Ampatzoglou, A. Chatzigeorgiou, S. Charalampidou, and P. Avgeriou, "The effect of gof design patterns on stability: A case study," *IEEE Transactions on Software Engineering*, vol. 41,

no. 8, pp. 781–802, Aug 2015.

[40] R. Vasa, J. G. Schneider, and O. Nierstrasz, "The inevitable stability of software change," in *2007 IEEE International Conference on Software Maintenance*, Oct 2007, pp. 4–13.

[41] A. P. S. de Moura, Y.-C. Lai, and A. E. Motter, "Signatures of small-world and scale-free properties in large computer programs," *Phys. Rev. E*, vol. 68, p. 017102, Jul 2003. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.68.017102

[42] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon, "Efficient sampling algorithm for estimating sub-graph concentrations and detecting network motifs," *Bioinformatics*, vol. 20, 2004.

[43] P. Tonella and G. Antoniol, "Inference of object-oriented design patterns," *Journal of Software Maintenance and Evolution: Research and Practice*, no. 13, pp. 309–330, 2001.

Table 6: Topological equivalent size-three motifs and their adjacency matrices

Motif ID	Motif structure	Adjacency matrix	Description	Topological equivalent motifs	Context	References
6		$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	Down - linked nodes	40, 192		
12		$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$	Three node chain	34, 66, 96, 132, 136	Food Webs	[7]
14		$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$	Down - linked node mutual dyad	42, 70, 168, 193, 224		
36		$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$	Up - linked nodes	72, 130		
38		$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}$	Feed forward loop	104, 134, 144, 194, 200	Software Systems, Gene regulation (transcriptions), Neurons, Electronic circuits, Object-oriented software	[15] [7][28]
46		$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$	Up - linked mutual dyad	198, 232	World Wide Web, Object-oriented software	[7] [28]
74		$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$	Up - linked node mutual dyad	76, 100, 138, 162, 164		
78		$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$	Two mutual dyads	170, 230		
98		$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$	Three-node feedback loop	140	Electronic Circuits	[7]
102		$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}$	Feed forward loop mutual dyad	116, 142, 172, 204, 226		
108		$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$	Down - linked mutual dyad	166, 202		
110		$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$	Feedback with two mutual dyads	174, 206, 230, 234, 236	World Wide Web	[7]
238		$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$	Fully connected triad		World Wide Web	[28]