e-Informatica Software Engineering Journal, Volume 8, Issue 1, 2014, pages: 7–26, DOI 10.5277/e-Inf140101

On Visual Assessment of Software Quality

Cezary Bartoszuk[*], Grzegorz Timoszuk[*], Robert Dąbrowski[*], Krzysztof Stencel[*]
[*]*Institute of Informatics, University of Warsaw*
cbart@students.mimuw.edu.pl, gtimoszuk@mimuw.edu.pl, r.dabrowski@mimuw.edu.pl, stencel@mimuw.edu.pl

Abstract

Development and maintenance of understandable and modifiable software is very challenging. Good system design and implementation requires strict discipline. The architecture of a project can sometimes be exceptionally difficult to grasp by developers. A project's documentation gets outdated in a matter of days. These problems can be addressed using software analysis and visualization tools. Incorporating such tools into the process of continuous integration provides a constant up-to-date view of the project as a whole and helps keeping track of what is going on in the project. In this article we describe an innovative method of software analysis and visualization using graph-based approach. The benefits of this approach are shown through experimental evaluation in visual assessment of software quality using a proof-of-concept implementation — the *Magnify* tool.

1. Introduction

Software engineering is concerned with development and maintenance of software systems. Properly engineered systems are reliable, efficient and robust. Ideally, they satisfy user requirements while their development and maintenance is affordable. In the past half-century computer scientists and software engineers have come up with numerous ideas for how to improve the discipline of software engineering. Edgser Dijkstra in his article [1] introduced structural programming which restricted imperative control flow to hierarchical structures instead of *ad-hoc* jumps. Computer programs written in this style were far more readable, easier to understand and reason about. Another improvement was the introduction of the object-oriented paradigm [2] as a formal programming concept in Simula 67. Other improvements in software engineering include e.g. engineering pipelines and software testing. In the early days software engineers perceived significant similarities between software and civil engineering. However, it has soon turned out that software differs from skyscrapers and bridges. The waterfall model [3] that resembles engineering practices was widely adopted as such, despite its original description actually suggesting a more agile approach. Contemporary development teams lean toward short iterations or so called sprints rather than fragile upfront designs. Short feedback loops allow customers' opinions to provide timely influence on software development. This improves the quality of the software delivery process.

In the late 1990s the idea of extreme programming (XP) emerged [4]. Its key points are straightforward: keep the code simple, review it frequently and test early and often. Among numerous techniques, XP introduced a test-driven approach to software development (today known as TDD). This approach encloses programming in a tight loop with three rules: (1) one cannot write production code unless there is a failing test; (2) when there is a failing test, one writes the simplest code for the test to pass; (3) after the test passes one refactors the code in order to remove all the duplicates and improve design.

This approach notably raised the quality of produced software and the stability of development processes [5].

The emergence of patterns and frameworks had a similar influence on architectures as design patterns and idioms did on programming. Unfortunately, there seems to be no way to test architectures in a similar way to testing code with TDD. Although we have the ideas of how to craft the architecture, we still lack the ways to either monitor the state of an architecture or enforce it. The problem skyrockets as software gains features without being refactored properly. Moreover, development teams change over time, work under time pressure with incomplete documentation and requirements that are subject to frequent changes. Multiple development technologies, programming languages and coding standards make this situation even more severe.

Our research pursues a new vision for management of software architecture. It is based on an architecture warehouse and software intelligence. *An architecture warehouse* is a repository of all software system and software process artifacts. Such a repository can capture architecture information which was previously only stored in design documents or simply in the minds of developers. *Software intelligence* is a tool-set for analysis and visualization of this repository's content [6–8]. That includes all tools able to extract useful information from the source code and other available artifacts (like version control history). All software system artifacts and all software engineering process artifacts being created during a software project are represented in the repository as vertices of a *graph*. Multiple edges of this graph represent various kinds of dependencies among those artifacts. The key aspects of software production like quality, predictability, automation and metrics are then expressed in a unified way using graph-based terms. The integration of source code artifacts and software process artifacts in a single model opens new possibilities. They include defining new metrics and qualities that take into account all architectural knowledge, not only the knowledge about source code. The state of software (the artifacts and their metrics) can be conveniently visualized on any level of abstraction required by software architects (i.e. functional level, package level, class or method level).

This article demonstrates a new idea and its proof-of-concept implementation – the tool *Magnify*. *Magnify* is focused on quick assessment and comprehension of software architectures. It visualizes relative importance of components, their quality and the density of their inter-connections. The importance is rendered using the size of symbols that depict components. It can be computed using multiple arbitrary metrics. In our experiments we utilized PageRank as a measure of component importance, which behaved well in practice. In order to visualize quality of a component we used colors, where as usual green denoted good quality, while red denoted poor quality. Again, there are multiple metrics that can be used to denote code quality. The examples presented in Section 7 use lines of code as the measure of quality. Such a simple metric may seem unreliable. However, it reflects the complexity of units of code (e.g. a class) and clearly indicates complicated entities. The initial results obtained from *Magnify* were presented in [9, 10].

The main contribution of this article when compared to our previous works [6–11] is the application of *Magnify* to a number of open-source projects and thorough analysis of the results. In our opinion *Magnify* can provide valuable insights into a project for architects and developers as discussed below.

We have considered numerous usage scenarios of *Magnify*. Assume newcomers approaching the project. They use this visualization to find starting points for their journey through development artifacts. They can even analyze whether it is worth joining a project. If the most important components are bright red and/or the coupling between those components is dreadfully dense, perhaps it is better not to embark this project. Architects can use the tool for everyday assessment of the system under their supervision. They can quickly notice e.g. (1) an unexpected emergence of a new important component, (2) a surprising degradation of a component, (3) a change in quality of a component (i.e. changing color from green to red), or (4) local or global

thickening of the web of dependencies among components.

The article is organized as follows. Section 2 addresses the related work. Section 3 recalls the graph-based model for representing architectural knowledge. Section 4 presents the method of quick assessment of software architecture. Section 5 presents usage scenarios of *Magnify*, and Section 6 shows its architecture. Section 7 demonstrates the application of *Magnify* to selected open-source projects. Section 8 concludes.

2. Related work

The idea described in this article has been contributed to by several existing approaches and practices. A unified approach to software systems and software processes has already been presented in [12]. Software systems were perceived as large, complex and intangible objects developed without a suitably visible, detailed and formal descriptions of how to proceed. It was suggested that software process should be included in software project as parts of programs with explicitly stated descriptions. The software architect should communicate with developers, customers and other managers through software process programs indicating steps that are to be taken in order to achieve product development or evolution goals. Nowadays, the process of architecture evolution is treated as an important issue that severely impacts software quality. There have been proposed formal languages to describe and validate architectures, e.g. architecture description language (ADL) [13]. In that sense, software process programs and programs written in ADLs would be yet another artifact in the graph recalled in this paper.

Multiple graph-based models have been proposed to reflect architectural facets, e.g. to represent architectural decisions and changes [14], to discover implicit knowledge from architecture change logs [15] or to support architecture analysis and tracing [16]. Graph-based models have also become helpful in UML model transformations, especially in model driven development (MDD) [17]. Automated transitions (e.g. from use cases to activity diagrams) have been considered [18] along with additional traceability that could be established through automated transformation. An approach to automatically generate activity diagrams from use cases while establishing traceability links has already been implemented (RAVEN) [19, 20].

As the system complexity increases the role of architectural knowledge also gains importance. There are multiple tools that support storing and analyzing that knowledge [21–24]. Architectural knowledge also influences modern development methodologies [25, 26]. It can be extended by data gathered during software execution [27]. The aspect of tracing architectural decision to requirements has been thoroughly investigated in [28–30]. An analysis of gathering, management and verification of architectural knowledge has been conducted and presented in [31]. Changes made in architecture management during last twenty years has been summarized in the survey [32].

There are also approaches to trace the architecture and its possible deterioration. The Structure101 tool [33] uses the Levelized Structured Map (LSM) to trace dependencies and to partition a system into layers. Another method called *Hyperlink Induced Topic Search* is used in [34] to evaluate object-oriented designs by link analysis. The method has been verified to identify God classes and reusable components. Furthermore, the idea of architectural constraints [35] in the form of constraint coupling can aid preventing architectural decay. However, the methodology and the tool *Magnify* described in this article are visual and not limited to layered architectures. Moreover, *Magnify* does not require adding new artifacts to a project (like constraints). Every software project can be evaluated by *Magnify* just as it is.

Visualization of software architecture has been a research goal for years. The tools like Bauhaus [36], Source Viewer 3D [37], Gevol [38], JIVE [39], evolution radar [40], code_smarm [41] and StarGate [42] are interesting attempts in visualization. However none of them simultaneously supports aggregation (e.g. package views),

drill-down, picturing the code quality and dependencies.

3. Graph model

In this Section we recall the theoretical model [7] for the unified representation of architectural knowledge. Such a model caters for the following key needs: (1) natural scalability, (2) abstraction from programming paradigms, languages, specification standards, testing approaches, etc, and (3) completeness, i.e. all software system and software process artifacts [12] are represented. The model is based on a directed labeled multigraph. A *software architecture graph* is an ordered triple $(\mathcal{V}, \mathcal{L}, \mathcal{E})$. $\mathcal{V}$ is the set of vertices that reflect all artifacts created during a software project. $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{L} \times \mathcal{V}$ is the set of directed labeled edges that represent dependencies (relationships) among those artifacts. $\mathcal{L}$ is the set of labels which qualify artifacts and their dependencies.

Vertices of the project graph are created when artifacts are produced during software development process. Vertices can represent parts of the source code (modules, classes, methods, tests), documents (requirements, use cases, change requests), coding guidelines, source codes in higher level languages (yacc grammars, web service specifications, XML schemata), configuration files, make files or build recipes, tickets in issue tracking systems etc. Vertices may be of different granularities (densities).

Vertices are subject to modifications during software development. It happens due to changes in requirements, implementation process, bug fixing or refactoring. Therefore, vertices must be versioned. Versions are recorded in labels containing version numbers (revisions) attached to vertices and edges. Thus, multiple vertices can exist for the same artifact in different version.

Example 3.1. *A method can be described by labels showing that it is a part of the source code (*code*); written in Java (*java*); its revision is 456 (*r:456*); it is* abstract *and* public*.*

3.1. Transformations

Transformations give the foundation for the *software intelligence* layer of the toolkit [6]. Our graph model is general and scalable as tested in practice [11]. However, in the case of a large project the model becomes too complex to be human-tractable as a whole. Software architects are interested both in an overall (top-level) picture and in particular (low-level) details. Selecting a specific subgraph is an example of a transformation (e.g. in a graph of methods with a call relation properly defined, the subgraph of methods that call the given method). Queries that compute such transformations are computationally inexpensive. Usually they only need to traverse a small fraction of the graph. Another important family of transformations are *transitions.* A transition maps a graph into a new graph and may introduce new vertices or edges, e.g. lifting the *dependency* relation from the level of classes (a class depends on another) to the level of packages. Further example of a transformation is a *map* that adapts a higher level of abstraction, e.g. hiding fields and methods while preserving class dependencies. Transformations can be combined.

Example 3.2. *For a given software graph $\mathcal{G} = (\mathcal{V}, \mathcal{L}, \mathcal{E})$ and a subset of its labels $\mathcal{L}' \subseteq \mathcal{L}$, the* filter *is a transformation $\mathcal{G}|_{\mathcal{L}'} = (\mathcal{V}', \mathcal{L}', \mathcal{E}')$ where $\mathcal{V}'$ and $\mathcal{E}'$ have a label in $\mathcal{L}'$.*

Example 3.3. *For a given software graph $\mathcal{G} = (\mathcal{V}, \mathcal{L}, \mathcal{E})$ and $t : \mathcal{L} \times \mathcal{L} \mapsto \mathcal{L}'$, the* closure *is the graph $\mathcal{G}^t = \{\mathcal{V}, \mathcal{L}', \mathcal{E}'\}$, where $\mathcal{E}'$ is the set of new edges resulting from the transitive closure of t calculated on pairs of neighboring edges from E.*

3.2. Metrics

The graph-based approach is in line with best practices for metrics [43, 44]. It allows the translation of existing metrics into graph terms [45]. It ensures that they can be efficiently calculated using graph algorithms. It also allows designing new metrics, e.g. such that integrate both soft-

ware system and software process artifacts. In our model metrics are specific transformations that map to the set of real numbers. For a given software graph $\mathcal{G} = (\mathcal{V}, \mathcal{L}, \mathcal{E})$, a *metric* is a transformation $m : \mathcal{G} \mapsto \mathcal{R}$ where $\mathcal{R}$ denotes real numbers and m can be effectively calculated by a graph algorithm on $\mathcal{G}$.

Example 3.4. *For a software graph G let CF be the* counting function $CF(n, \eta_1, \eta_2, \eta_3) = \#\{m \in \mathcal{V} \mid type(n) \ni \eta_1 \wedge type(m) \ni \eta_2 \wedge \exists e \in \mathcal{E}\colon source(e) = n \wedge target(e) = m \wedge type(e) = \eta_3\}$, *where $n \in \mathcal{V}$, $\eta_1, \eta_2, \eta_3 \in \mathcal{L}$. For a node n with a label in set η_1, CF counts the number of nodes m with a label in set η_2 such that there is an edge e of label η_3 from n to m. CF can be implemented on G in $O(|G|)$ time.*

Example 3.5. *Let* NOC *(Number Of Children) denote a metric that counts the number of direct subclasses. Calculating such metric in graph-based model reduces to filtering and counting neighbors. It can be done quickly, i.e. in $O(|G|)$ time. Using the counting function* NOC *is implemented simply as: $NOC(c) = CF(c, class, class, inherits)$.*

Example 3.6. *Let* WMC *(Weighted Method per Class) denote a metric that counts $\sum_{i=1}^{n} c_i$ where c_i is the complexity of the i-th method in the class. If each method has the same complexity,* WMC *is just the number of methods in the class. Using the* counting function *the number of methods in a class is implemented simply as: $WMC(c) = CF(c, class, method, contains)$.*

Graph metrics depend only on the graph's structure. They are independent of any programming language. Hence storing and integrating all architectural knowledge in one place facilitates tracing not only dependencies in the source code but also among documentation and meta-models. This opportunity gives rise to new graph-defined metrics concerned with software processes.

Example 3.7. *Let* CHC *(Cohesion of Classes) denote a metric that counts the number of strongly connected components of this graph. A software is cohesive if this metric is* 1. *In the graph-based model it is computed quickly, in time $O(|G|)$.*

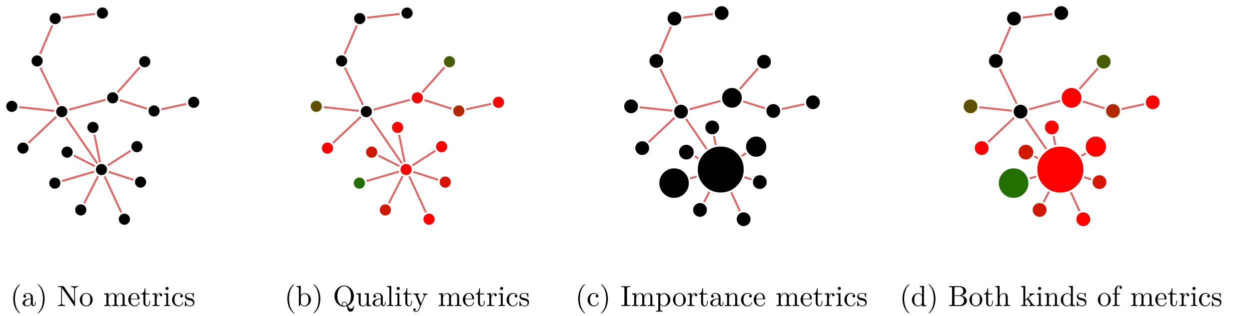
4. Software analysis method

Our method uses software architecture graphs (see Section 3). Its goal is quick assessment and comprehension of software projects. Assume a software architecture graph $\mathcal{G} = (\mathcal{V}, \mathcal{L}, \mathcal{E})$ such that $\mathcal{L}|_V = \{package, class\}$ and $\mathcal{L}|_E = \{contains, calls, imports\}$. A package *contains* classes and packages. A package *imports* a package. A class *calls* a class. We also apply a *transition* of $\mathcal{G}$ that combines the relation *contains* of *packages* and *classes* and the relation *calls* between classes. Its result is the relation *calls* among packages.

4.1. Visualization

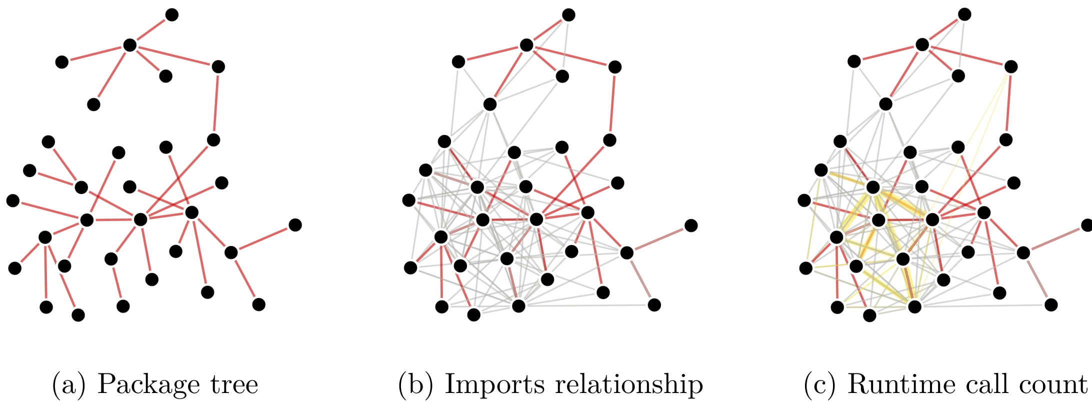
A quick assessment and comprehension of a software project can be done by a visualisation of the two dimensions: (1) *importance* and (2) *quality* of software artifacts. Following the research on warehousing and analysis of architectural knowledge [6, 7], we visualize the software in the form of a planar representation of the directed multigraph of software artifacts and their relations. We render the two dimensions using size and color. The *size* of a node depicts its artifact's *importance.* The *color* of a node shows its artifact's *quality.* Intuitively, a *big node* denotes an important artifact, while a *small node* denotes an unimportant one. A *green node* denotes an artifact of good quality, while a *red node* denotes an artifact of poor quality. An artifact depicted as a *big red node* should gain attention of software architects and engineers because of its high importance and poor quality. Figure 1 shows basic examples.

As defined in Section 3, the graph-based model embraces all types of artifacts that occur in a software project and all types of their relations. Those include non-software artifacts like use cases or artifacts related to the software development process and additional attributes for graph vertices and edges. We can e.g. enrich the *calls* relation with the attribute *call count* collected during a runtime analysis [46]. This kind of data can be obtained using frameworks



(a) No metrics (b) Quality metrics (c) Importance metrics (d) Both kinds of metrics

Figure 1. Software artifacts – their importance and quality



(a) Package tree (b) Imports relationship (c) Runtime call count

Figure 2. Relationships between software artifacts – static and dynamic

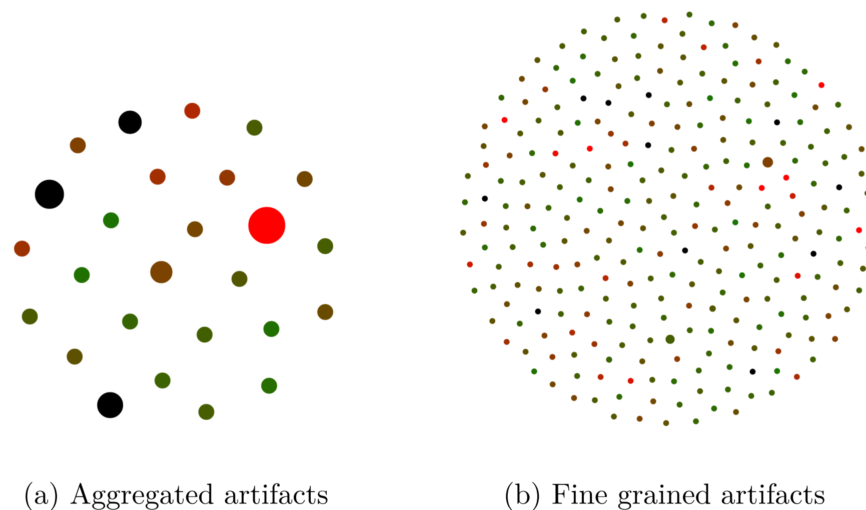
like Kieker [27]. Such dimension as *call count* can be depicted by thickness of graph edges. A *thick edge* denotes frequent calls and a *thin edge* denotes rare calls (see Figure 2c).

4.2. Analysis

In this section we assume a software project with the following properties. (1) Static data, like the source code, has been uploaded into its software architecture warehouse. (2) Dynamic data, like the runtime log of procedure calls, has been uploaded into its software architecture warehouse. (3) These data have been preprocessed, in particular different project metrics have been calculated. This allows the preparation of a visualisation of this software project that facilitates its interesting multi-dimensional analysis. Let us review the key points of the presented approach.

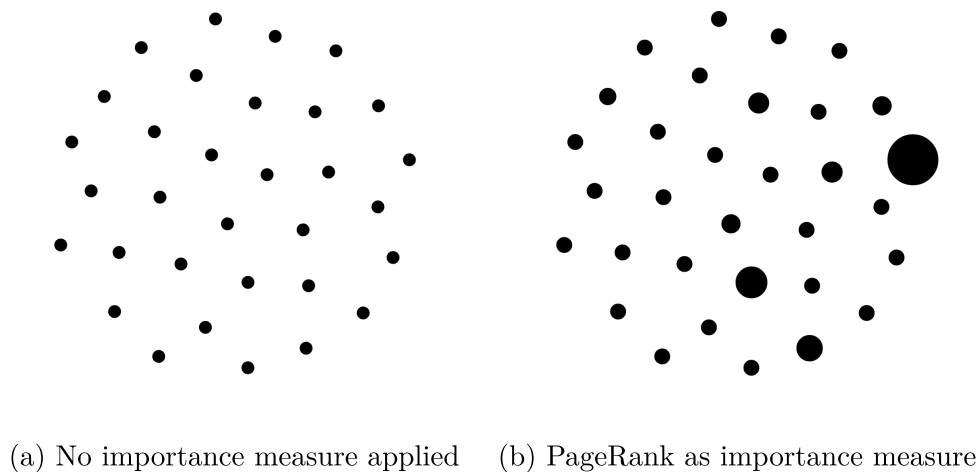
We have to select artifacts to be depicted as nodes of the graph. The size of this collection is first of all determined by the **abstraction level** of the assessment. Then further filters or transformations can be applied (see Section 3). Figure 3 shows two abstraction levels. Figure 3a shows a smaller collection of top-level packages. Figure 3b shows a bigger collection of low-level classes.

There can be multiple intuitions behind the definition of the **importance** of artifacts, e.g. the amount of work needed to adjust the rest of the system if this part of code gets changed. Consequently, there can be different algorithms that implement those intuitions with different semantics. In particular *PageRank* [47] assigns higher importance to more *popular* nodes. The more edges point a node, the higher is its rank. Such measure properly reflects the practical importance of software artifacts. Figure 4 shows sample visualizations. Figure 4a does not show importance, while Figure 4b has big nodes for



(a) Aggregated artifacts (b) Fine grained artifacts

Figure 3. The importance and the quality at different levels of abstraction



(a) No importance measure applied (b) PageRank as importance measure

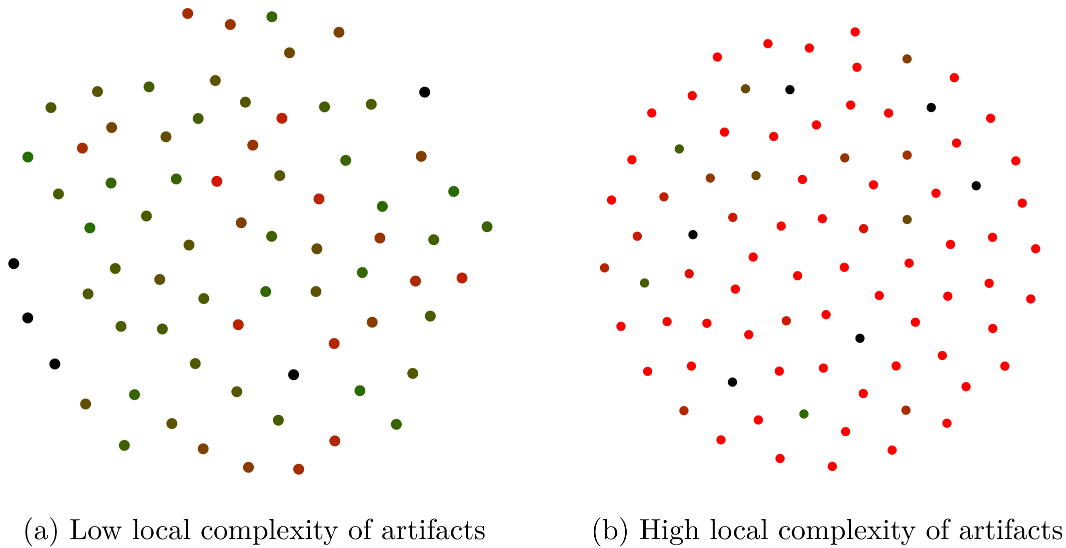
Figure 4. Two visualizations of nodes – with and without importance shown

important packages and small nodes for less important packages.

There can also be multiple intuitions behind the definition of the **quality** of artifacts. One possible interpretation is the local complexity for which there are numerous possible metrics. One of the most popular is *Cyclomatic Complexity* [48]. Figure 5 shows quality of artifacts visualized for two different projects. The one from Figure 5a seems to have low local complexity as most artifacts are green-brownish. The one from Figure 5b has few artifacts with reasonable local complexity.

When the measure of quality is also a measure of complexity, it does not have to be independent from the PageRank. The more complex the class, the more links it usually has. Those links tend to raise the PageRank. In our experiments (see Section 7), we have not observed this dependency. The quality measure has been the average number of lines of code per class. It is obviously also a complexity measure. However, the pictures generated by *Magnify* do not confirm its substantial dependency on PageRank.

Architects usually start depicting a system at the top-level where vertices are packages. For most of the software projects they are granular enough and their amount remains comprehensible for a human. When an architect moves to a



(a) Low local complexity of artifacts (b) High local complexity of artifacts

Figure 5. The quality of artifacts - low vs. high local complexity

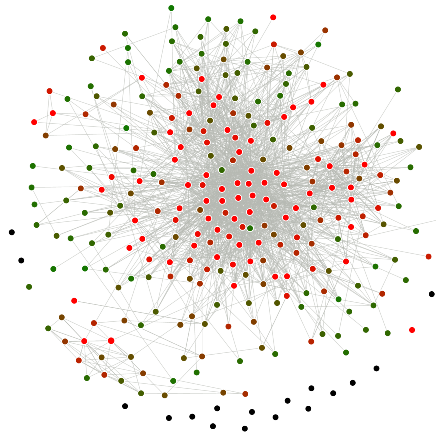


Figure 6. The complexity of relations may require applying model transformations.

lower abstraction level where nodes are classes, the picture gets complicated. In such case, interesting edges are of several kinds. They are: (1) a *dependency* of a class on another class, if the former knows about the latter. (2) an *inclusion* of a method in a class, a class in a package, a package in its parent package, (3) a *call* between two classes, if any method of the first class calls the second class. Moreover, some of dependencies cannot be observed just by processing source code at the design time. Modern programming languages provide means for dynamic calls. Thus, runtime analysis is required. At some abstraction levels a dense net of dependencies may occur (see Figure 6). For such cases, our model offers multiple graph transformations to support software architects and engineers, like filtering, mapping, zooming etc.

5. Magnify

Magnify is a proof-of-concept implementation of the ideas presented in this article. It visualizes software projects as graphs. The project including its source code can be downloaded from `https://github.com/cbart/magnify`. We start from browsing through potential user groups that can be interested in using *Mag-*

nify. The following sections describe real world situations where our tool can prove useful.

5.1. Software architects

Nowadays well managed software teams have a sophisticated infrastructure that aids efficient development and reduces risks. Tools and techniques used in a modern software project should include e.g.: version control, unit testing, continuous integration, code review, code analysis tools including copy paste detectors, complexity metrics, bug finders, automatic deployment, stand-ups, short iterations or sprints, planning meetings and retrospectives. Our tool fits in this scheme as a code analysis tool that can be run frequently (e.g. for each revision, hour or day).

Software architects can use *Magnify* to obtain an up-to-date holistic view that shows how the overall development is proceeding in terms of emerging code artifacts and dependencies. The architects can quickly notice if recent changes break e.g. software modularity or other architectural ideas. *Magnify* can also be used during retrospectives. It allows a scrum master to visualize different revisions of software. The team can quickly see what was the effect of the given sprint of their work.

5.2. Software engineers

Software teams can use *Magnify* to continuously analyze their own software in order to improve its quality. They can also use *Magnify* as a tool for analyzing foreign projects. Assume a software engineer wants to join a new project. Typically, he/she would contact the development team and check what programming languages, tools, libraries, techniques and practices they use. If he/she wanted to check the quality of the system under development, he/she would check its test coverage, run a static code analysis and observe the software in runtime environment. *Magnify* offers a view of a project from a unified high-level perspective that gives all those valuable insights.

Consider a perspective where an open-source solution is incorporated into the system being developed. The usual approach is to introduce abstractions between this system and the third party library or framework. This significantly increases the flexibility. The development team can also upgrade the third party software and only reimplement a façade to make everything work. Sometimes, though, this is not an option. When an open source software does not provide all the functionality that is needed, there exist only few possible solutions. The team can introduce the needed changes into the next versions of the open source itself. Sometimes the ideas of the team do not match the concept of the library's architect. Then an implementation of such changes in the library's main branch becomes a management problem. Even if the changes get allowed, their implementation and review gets time consuming. The other way is to fork the open source project and develop the needed changes in house. One of the consequences of choosing this path is the lack of support from the library's authors. In this case the team might want to examine the third party software before they start contributing. As described in Section 5.1 they can use various tools to investigate the quality. Among those tools *Magnify* provides a starting-point view of the project, depicting it as-is in a unified, high-level perspective. In Section 7 we present examples of software project properties that are well visualized using *Magnify*.

5.3. Computer scientists

Magnify can also be used for scientific software inspection. Thanks to flexibility of the graph model presented in Section 3, *Magnify* is an effective software analysis framework. Scientists that analyze software can easily implement graph transformations and custom code metrics. The graph model and the architecture described in Section 6 allow using a plethora of well known graph algorithms in their research.

6. Architecture

Magnify is a JVM application with web interface. In this Section we describe the architecture of our tool.

Visualization (SVG)	Control (JavaScript, DOM)
Presentation (d3.js)	
Data boundary (HTTP, REST, JSON)	
HTTP server (Scala, Play)	
Graph views (Scala, Gremlin)	
Graph database (Tinkerpop, Blueprints, Neo4j)	

Figure 7. The architecture of the visualisation part of *Magnify*

Nowadays there are numerous storage technologies available. For the last 30 years the database community has been dominated by relational databases with popular database management systems like Oracle, Postgres, MySQL, Microsoft SQL Server or SQLite. During that time so called *SQL databases* were the default choice as persistence layers. Most recently the movement of the *NoSQL* emerged. It is focused on non-relational (sometimes even schema-less) database technologies including column-oriented database management systems (e.g. Google's BigTable and Apache HBase), key-value stores (e.g. Riak and Redis), document stores (e.g. MongoDB and CouchDB) and graph databases (e.g. Neo4j and OrientDB). When implementing *Magnify* we considered multiple options for the storage layer. The graph databases always seemed the most in line with our graph model described in Section 3. Tinkerpop Blueprints is a standard model for working with graph databases on the JVM. With its flexible query language Gremlin and a simple graph model, it made easy to implement needed graph transformations. Thanks to Blueprints Graph implementations we were able to use ready implementations of needed algorithms. For example, we use a PageRank implementation from the *Java Universal Network/Graph Framework* (or JUNG) thanks to the provided Blueprints JUNG implementation.

The main feature of *Magnify* is visualisation of the software graph. There are abundant technical possibilities to achieve such goal in a browser. For two-dimensional diagrams that are composed of simple shapes, SVG (scalable vector graphics) seems to be the simplest solution. Elements of an embedded SVG image are plain old XML tags and thus belong to the DOM. Thanks to that they can be manipulated and can react to DOM events (click, hover, etc.) such as any other parts of HTML page.

In *Magnify* we used a library called *d3.js*, i.e. a multi-purpose visualisation framework. It offers tools for creating, manipulation of SVG graphics and reacting to DOM events. In *Magnify* we used a custom force directed graph. We use Force Atlas as our layout algorithm with attracting force on edges, repulsive charges and gravity on graph nodes. In practice it has proved to be a fine way to visualize software graph on a plain.

There are disparate data formats to represent a property graph model presented in Section 3. One of the most popular is the *Graph XML Exchange Format* (*GEXF*) format. The schema of *GEXF* is extensible enough to contain all the required properties of nodes and edges. It can be read by popular graph manipulation tools like *Gephi*. At the moment of writing *Magnify* supports graph import and export in *JSON* format. This was the most convenient format for integration with other tools in our research.

7. Experimental evaluation

In this Section we show the results of applying *Magnify* to the following eight open-source projects: Apache Maven 3.0.4, JLoXiM rev.2580, Weka 3.5.7, Spring Context 3.2.2, JUnit 4.10, Cyclos 3.7, Play 1.2.5, Apache Karaf 3.0.0 RC1. The projects significantly vary in size, quality, purpose and design. *JLoXiM* is a research project developed by students. It was a case-study in our previous experiments [11]. The remaining seven projects are well-recognized systems, frameworks and libraries.

For each system we present its visualisation created by *Magnify* and sample conclusions

drawn from this view. Wherever a listed conclusion concerns only a part of the visualization, we add an oval to the figure in order to indicate the subject area. We label these ovals with identifiers of observations.

7.1. Apache Maven 3.0.4

Apache Maven is a build automation tool. It serves a similar purpose to *Apache Ant*. It compiles, packages and deploys projects. Maven supports dependency management. It can download external modules and plugins from remote repositories like the *Maven 2 Central Repository*. Figure 8 shows its visualizations.

Observation 7.1. *`org.apache.maven.model` is well encapsulated.*

Let us focus on the group of packages on top of Figure 8b. When we point its center with the mouse, a tooltip will inform that the name of this package is `org.apache.maven.model`. Only four packages are visible outside this group: `building`, `io`, `plugin` and `resolution`. That means that all the other artifacts inside `org.apache.maven.model` can change without affecting the rest of the system. In fact when you take a look at the structure of Maven subprojects, you can see these two: `maven-model` and `maven-model-builder`. The subproject `maven-model` contains mostly tests and only one public non-test class. The subproject `maven-model-builder` contains all the other classes under the `model` package. Thus, in case of `org.apache.maven.model` subprojects the directory structure properly reflects underlying code dependencies.

Observation 7.2. *Dependencies around `org.apache.maven.artefact` form a dense network.*

There is an entanglement on the bottom side of the picture around `org.apache.maven.artefact`. The gray area in Figure 8a shows substantial amount of dependencies. This means that the code in this part of the project is tightly coupled. Therefore, if some pieces change, numerous other items will be affected. Fixing this tight coupling is not easy as it requires diving deeper into the code and refactoring the design of how the classes cooperate.

Observation 7.3. *The overall local complexity is satisfactory.*

Apart from the package tree, Figure 8c shows both the importance and local complexity of nodes. Most of the nodes are green-brownish. Thus, the overall quality of classes is satisfactory.

7.2. JLoXiM, revision 2580

JLoXiM is an experimental semi-structured database management system. It is developed by a team of students that is subject to frequent changes. This makes it an interesting case for analysis of architectural changes [11]. Figure 9 presents the visualisation of this system by *Magnify.*

Observation 7.4. *Parts of JLoXiM have modular structure and are well encapsulated.*

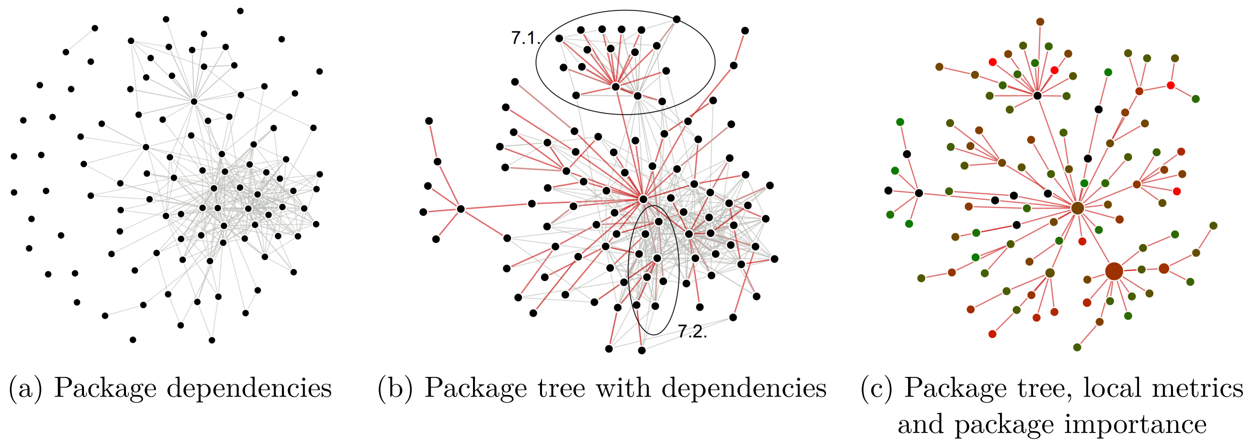
When we look at Figure 9, we can graphically divide the system into two parts. The bottom part has dense dependencies. The top part contains few aggregates of packages. The groups of packages on top have numerous internal edges, i.e. dependencies inside the aggregate. However, the dependencies between the groups are notably reduced.

The top part seems well designed from the architectural point of view. Low level of density between the aggregates indicates that they are loosely coupled. Thus, all the pieces are easily exchangeable. This substantially increases the ease of development and the flexibility of the resulting solution.

On the other hand the groups themselves are far more dense inside than outside. This means that there are classes that are closely related. Therefore, one could form modules that would be both easily interchangeable and easy to understand by developers. Unfortunately, they are not always packaged as the package dependencies would suggest.

Observation 7.5. *JLoXiM is not well packaged.*

In Figure 9 red edges form the package tree. In several parts of *JLoXiM* the dependencies go against packaging, i.e. there are sections of the package tree that are highly coupled even though they are not packaged together. When *Magnify* applies more attractive power to depen-



(a) Package dependencies (b) Package tree with dependencies (c) Package tree, local metrics and package importance

Figure 8. The visualisation of Maven 3.0.4 using *Magnify*

dencies, the package tree itself looks like a tangle. Since there are abundant dependencies on pieces of code that are not close in the package tree, browsing the code is particularly difficult. Tracking the flow requires jumping back and forth from one package to another. A way to avoid that inherent complexity is to repackage classes in a more natural manner that embraces their dependencies.

Observation 7.6. *There are no God Modules in JLoXiM.*

A *God Module* is a piece of code that contains too many responsibilities and seems to do everything. In *Magnify* its size skyrockets compared to the other nodes. Besides few slightly bigger nodes, packages of JLoXiM are more or less of the same size. Therefore, the architecture is balanced.

Indeed, when one inspects the code with text processing tools, it becomes obvious that besides the five most often imported classes (that are value objects), all the others are imported less than 100 times each. This is not too much for approximately 2100 classes in the whole project.

Observation 7.7. *Less than a half of JLoXiM code is touched at runtime by the test suite.*

Besides dependencies (gray edges) and package tree (brown edges) Figure 9 presents also yellow edges which visualize the control flow. The thicker is a yellow edge the more flow went from one package to another during the runtime monitoring session. This kind of experiment performed on different environments can yield interesting results. One could monitor how control flow passes in a production environment. This kind of monitoring brings a significant performance overhead. On the other hand plugging it into only small percent of production instances should not affect the overall performance too much. However, it can produce a significant amount of important data. Another scenario might be capturing call count during running an acceptance test suite. For example, if a team wants to introduce continuous deployment in their release and drifts towards fully automatic shipping, then their acceptance test suite will have to embrace most of the code. In this case visualizing call count can prove interesting in two ways. (1) It can help identify dead flows that are not needed any more and have become clutter over the history of this system development. (2) It can point out important flows that are not covered by acceptance test scenarios.

7.3. Weka 3.5.7

Weka is a collection of machine learning algorithms. It is used mainly for data mining and contains tools for classification, regression, clustering, association rules and more. Figure 10 presents visualizations of Weka packages using *Magnify*.

Observation 7.8. *Weka classes are large and complex.*

The first things to notice at Figure 10b are red nodes. The red color of nodes indicates that per-package average local complexity of Weka

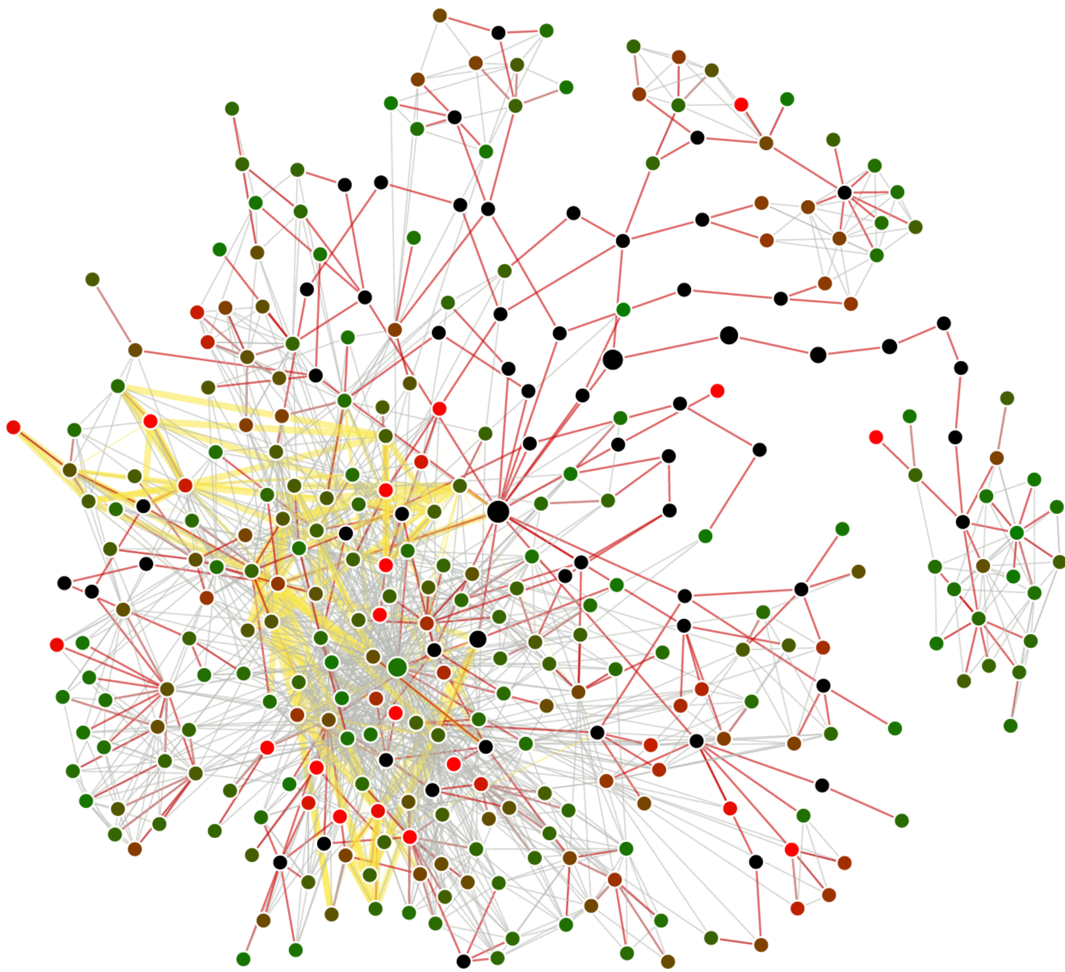


Figure 9. The visualisation of JLoXiM r2580 using *Magnify*.

classes is notably high. Thus the code is difficult to grasp and maintain. Fortunately the problem of local complexity is easy to fix. Modern tools including IDEs provide numerous methods that help moderating local complexity of classes. With series of refactorings one could significantly reduce this inherent complexity.

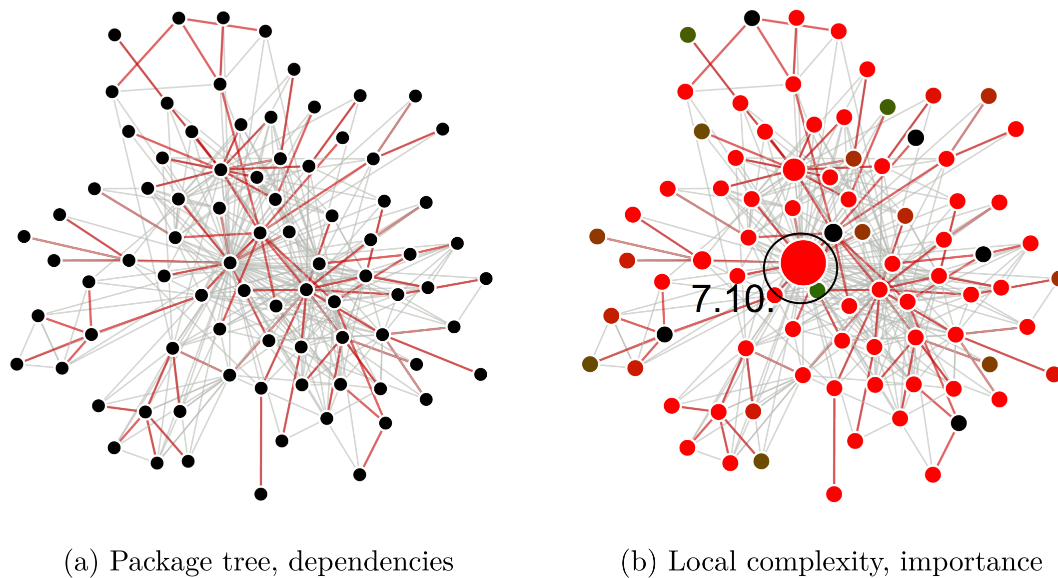
Observation 7.9. *Weka does not have modular architecture.*

Dependencies between packages do not seem to form any modular patterns. The graph is relatively dense for such a small project. Compared to the previous flaw this one is far more difficult to fix. Repackaging and module formation usually requires deep understanding of the system under refactoring as well as the domain it works in.

Observation 7.10. `weka.core` *might be a God Module.*

The package node `weka.core` seems to be far bigger than all the others. Moreover, it holds a significant number of dependencies and seems to be the central place of the system. The bugs in this part are potentially destructive.

The package `weka.core` contains 80 classes itself, which is far too much. We looked in more depth at `weka.core.Utils`. This class is approximately 2000 lines long. It contains unrelated utility static methods. To our surprise the quality of the underlying code is fairly good. The methods themselves are short and concise, but there are too many of them. A simple refactoring that will significantly improve this structure consists in extracting classes containing cohesive methods



(a) Package tree, dependencies (b) Local complexity, importance

Figure 10. The visualisation of Weka 3.5.7 using *Magnify*.

like string manipulation, statistics, comparisons and so on.

7.4. Spring context 3.2.2

Spring is one of the most popular enterprise application frameworks in Java community. It provides an infrastructure for dependency injection, cache, transactions, database access and many more. Figure 11 shows how it looks in *Magnify*.

Observation 7.11. *Spring is well designed.*

The dependency graph is noteworthy sparse with a few dependencies between packages. Thus, the overall coupling in the code is low and/or the packages are self-dependent.

Observation 7.12. *Packages are of equal importance.*

The only packages that are indicated as important are empty vendor packages: the root package, `org` and `org.springframework`. These packages are not used to store code. They just form a namespace for the project. All packages that contain any classes are of same importance. This resembles a well balanced piece of software.

Observation 7.13. *The overall quality of code is satisfactory.*

There are no bright red packages in the picture. Most of nodes are colored from green to red-brownish. This means that on average classes are small in most of packages. With smaller classes it is far easier for developers to get to know the code. If a class is small enough, even if the code inside is complex, the idea behind it will be easy to understand.

7.5. JUnit 4.10

JUnit is a unit testing framework for Java. Started by Kent Beck and Erich Gamma it gained popularity and it is still helping test drive modern Java projects. Figure 12 shows its visualizations.

Observation 7.14. *Overall code quality is good.*

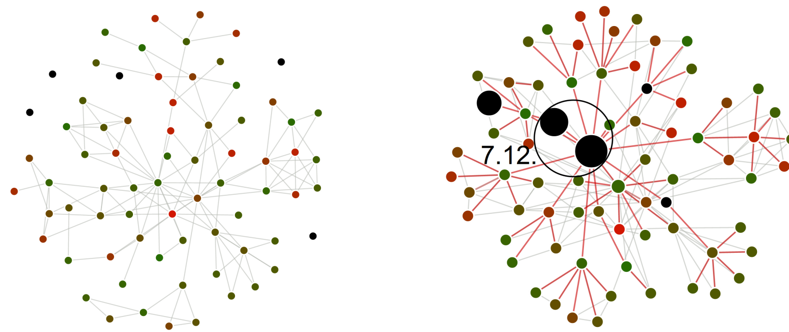
We can see that all important packages are green and the web of dependencies is manageable.

Observation 7.15. *Not all parts of JUnit were executed during our test example.*

The runtime data visible in Figure 12c are call counts inside JUnit library gathered while running one of our test suites. The yellow edges do not touch all the packages of this small library. In this particular case the reason might be that our test case did not use all the features JUnit has to offer.

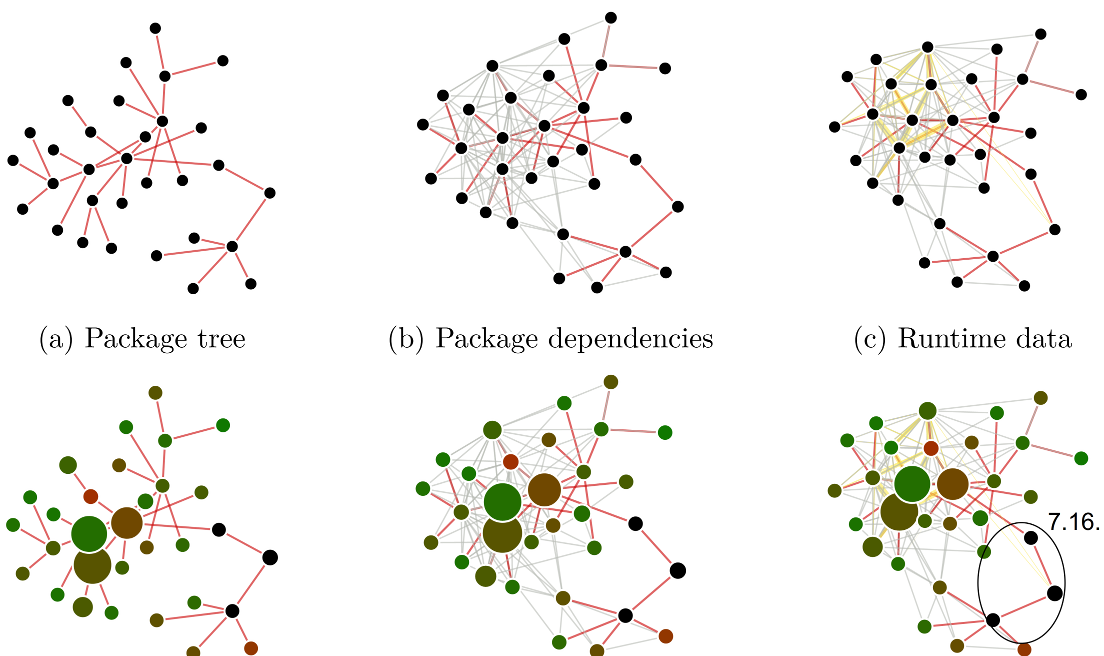
Observation 7.16. *Some runtime dependencies are not in line with static dependencies.*

One can also spot one peculiar thing. Near the bottom right corner of Figure 12f we can



(a) Dependencies and complexity (b) Package tree and importance

Figure 11. The visualisation of Spring context 3.2.2 using *Magnify*.



(a) Package tree (b) Package dependencies (c) Runtime data

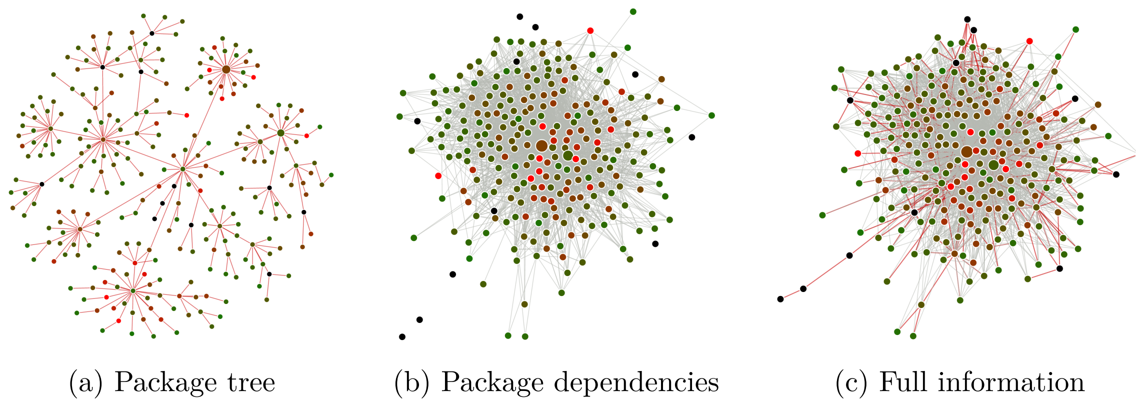
(d) Package tree, metrics (e) Package dependencies, metrics (f) Runtime data, metrics

Figure 12. JUnit 4.10 visualized with *Magnify*

see three black nodes. These are (from top to bottom) the `org` package, the root package and the `junit` package. There are two thin runtime flow edges adjacent with the root package. Our first thought when analyzing this visualisation was that there are some classes in the root package that are accessed via the reflection. A deeper investigation had proven that these edges show the use of *dynamic proxies* which get compiled into classes that end up in the root package.

7.6. Cyclos 3.7

Cyclos is a complete on-line payment system. It also offers numerous additional modules such as e-commerce or communication tools. The project allows local banks to offer banking services that can simulate local trade and development. Cyclos is published under the GPL license. Figure 13 presents visualizations of this system in the *Magnify* tool.



(a) Package tree (b) Package dependencies (c) Full information

Figure 13. The visualization of Cyclos 3.7 using *Magnify*.

Observation 7.17. *The network of dependencies is exceptionally dense.*

The experience shows that software systems with abundant inter-dependencies tend to be difficult in comprehension, maintenance and development. Such systems are also exceptionally fragile. In dense dependency networks a software engineer struggling to understand a piece of code must read through several other pieces this piece is dependent on. Cyclos is fragile because a bug in one part of code affects multiple other parts. Furthermore, a modification, an improvement or refactoring of a single piece of code causes copious additional changes since its neighborhood is always big.

Observation 7.18. *The local complexity of classes is manageable.*

Figure 13b shows few packages in which classes are big on average. That means that overall complexity of the classes themselves is acceptable.

Observation 7.19. *Cyclos should be split into cooperating subsystems.*

Cyclos is a profound example of a system that should be split into orchestrated group of communicating systems. This kind of refactoring would significantly improve the quality of this software itself as well as the costs of further development. In our opinion, the introduction of the Service Oriented Architecture or the Microkernel with Services would benefit the developing team. This way system parts would have clearly defined boundaries, e.g. in the form of RPC interfaces. Since dealing with separate services makes it more difficult to depend directly on implementation details, it discourages high coupling between services. As long as services are loosely coupled and small, the code inside them can be fairly complex, since rewriting a single service from scratch is significantly less costly than rewriting the whole system.

7.7. Play 1.2.5

Play is a popular Scala and Java web framework. It is built on a lightweight, stateless and web friendly architecture. Play is heavily influenced by dynamic language web frameworks like Rails and Django. That makes a simpler development environment when compared to other Java platforms like JEE or Struts. Figure 14 shows visualisation of Play using *Magnify*.

Observation 7.20. *The package structure is flat.*

Figure 14 shows a small project with fair amount of dependencies. The height of the package tree is small. Unlike classic JVM package trees this kind of flat package structure is typical for dynamic languages. The packaging approach the Play team has taken emphasizes the influence by popular rapid application development web frameworks from the family of dynamic languages.

Observation 7.21. *The package* `play` *seems like a do-it-all framework façade.*

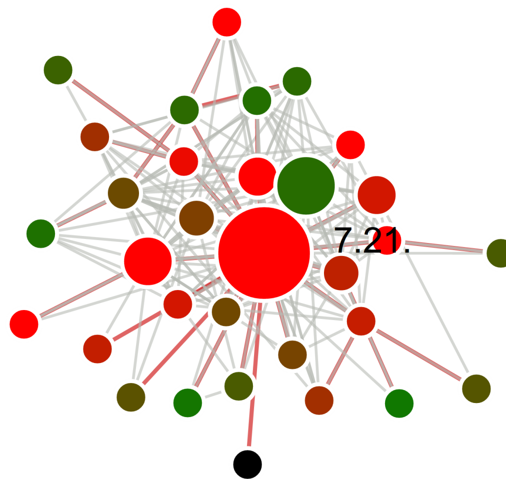


Figure 14. The visualization of Play 1.2.5 using *Magnify*.

The biggest node corresponds to the project root package `play`. Bright red color reveals potentially high complexity of classes inside. It is customary in dynamic languages to expose most of library or framework functionality through few classes contained in a single name space. Among other things, beginners can more easily find all the needed endpoints. For example, in Scala they can simply `import play._` and have access to all the features they need.

7.8. Apache Karaf 3.0.0 RC1

Apache Karaf is a small OSGi container in which various components and applications can be deployed. Karaf supports hot deployment of OSGi bundles, native operating system integration and more. Figure 15 shows *Magnify* visualizations of Apache Karaf.

Observation 7.22. *Apache Karaf is well packaged.*

Even though Karaf is split into plentiful packages, the number of dependencies is small. Most subtrees of package hierarchy have only a single dependency on the rest of the system. That implies a well packaged system.

Observation 7.23. *Local code quality is fair.*

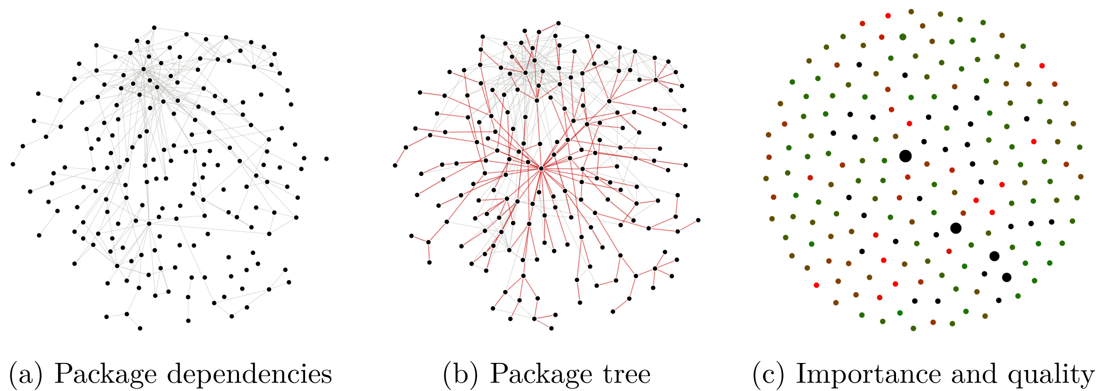
Figure 15 shows that overall code quality in Karaf is good. There are only few packages where average class size is alarming. The only refactoring we can suggest is to encapsulate subpackages of `org.apache.karaf.shell` which tend to spread a web of dependencies in the top part of the picture.

8. Conclusions

In this article we described the tool *Magnify*. We explained how architects could use *Magnify* in order to quickly comprehend and assess software. The idea is to automatically generate a visualisation of the software such that architects can instantly see the importance and the quality of software components. They can do it at the level of abstraction they require.

We have also performed experimental evaluation of our approach. The experiments have proven that a sparse software graph and almost uniformly distributed node sizes mean a proper modular architecture. On the other hand, one node dominating others in size might also mean a shared kernel architecture, where other functionalities are implemented as services floating around the kernel.

Magnify is a general tool that can adopt other quality metrics and importance estimates. Although PageRank as the algorithm to compute importance have proven to be effective in practical applications, its adequacy can be questioned. For example, a common technique for encapsulating a module in an object-oriented language involves depending on a module's interfaces and obtaining instances via a façade. PageRank im-



(a) Package dependencies (b) Package tree (c) Importance and quality

Figure 15. The visualisation of Karaf 3.0.0-RC1 using *Magnify*.

portance of the façade will be significantly higher than importance of implementation classes. This usually is a poor reflection of the real importance.

Magnify can be extended in disparate directions. Currently *Magnify* supports only Java. Adding support for other programming languages requires registering a new parser. Its duty is to analyze source files and add specific nodes and their relations into the graph database. Since the graph-based representation of the source code is language agnostic, all the analysis done inside *Magnify* will work equally well for any language with notions of packages, classes and methods.

Furthermore, even though certain local complexity measures might depend on a programming language, most of them do not. The cyclomatic complexity that takes into account execution paths can be computed in the same way for most programming languages. Moreover, most languages use the same keywords for branching and loops. Thanks to that and the syntactic nature of the cyclomatic complexity one can write an implementation that works well with most of the popular programming languages.

Magnify is implemented using standards for representation, storage and visualisation of graphs, e.g. Blueprints API or the GEXF graph format. Measures of importance of a node depend only on the used graph model. Thus, any algorithm working on those standard graph technologies will do.

References

[1] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful," *Commun. ACM*, Vol. 11, No. 3, 1968, pp. 147–148.

[2] J. McCarthy, M. I. of Technology. Computation Center, and M. I. of Technology. Research Laboratory of Electronics, *Lisp one five programmer's manual*. Massachusetts Institute of Technology, 1965. [Online]. http://books.google.pl/books?id=68j6lEJjMQwC

[3] W. Royce, "Managing the development of large software systems: Concepts and techniques," in *WESCOM*, 1970.

[4] K. Beck, "Embracing change with extreme programming," *IEEE Computer*, Vol. 32, No. 10, 1999, pp. 70–77.

[5] R. Kaufmann and D. Janzen, "Implications of test-driven development: a pilot study," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 298–299. [Online]. http://doi.acm.org/10.1145/949344.949421

[6] R. Dąbrowski, "On architecture warehouses and software intelligence," in *FGIT*, ser. Lecture Notes in Computer Science, T.-H. Kim, Y.-H. Lee, and W.-C. Fang, Eds., Vol. 7709. Springer, 2012, pp. 251–262.

[7] R. Dąbrowski, K. Stencel, and G. Timoszuk, "Software is a directed multigraph," in *ECSA*, ser. Lecture Notes in Computer Science, I. Crnkovic, V. Gruhn, and M. Book, Eds., Vol. 6903. Springer, 2011, pp. 360–369.

[8] R. Dąbrowski, G. Timoszuk, and K. Stencel, "One graph to rule them all software measurement and management," *Fundam. Inform.*, Vol. 128, No. 1-2, 2013, pp. 47–63.

[9] C. Bartoszuk, G. Timoszuk, R. Dąbrowski, and K. Stencel, "Magnify - a new tool for software visualization," in *FedCSIS*, M. Ganzha, L. A. Maciaszek, and M. Paprzycki, Eds., 2013, pp. 1473–1476.

[10] C. Bartoszuk, R. Dąbrowski, K. Stencel, and G. Timoszuk, "On quick comprehension and assessment of software," in *CompSysTech*, B. Rachev and A. Smrikarov, Eds. ACM, 2013, pp. 161–168.

[11] R. Dąbrowski, K. Stencel, and G. Timoszuk, "Improving software quality by improving architecture management," in *CompSysTech*, B. Rachev and A. Smrikarov, Eds. ACM, 2012, pp. 208–215.

[12] L. J. Osterweil, "Software processes are software too," in *ICSE*, W. E. Riddle, R. M. Balzer, and K. Kishida, Eds. ACM Press, 1987, pp. 2–13.

[13] M. T. T. That, S. Sadou, and F. Oquendo, "Using architectural patterns to define architectural decisions," in *WICSA/ECSA*, T. Männistö, A. M. Babar, C. E. Cuesta, and J. Savolainen, Eds. IEEE, 2012, pp. 196–200.

[14] M. Wermelinger, A. Lopes, and J. L. Fiadeiro, "A graph based architectural (re)configuration language," in *ESEC / SIGSOFT FSE*, 2001, pp. 21–32.

[15] A. Tang, P. Liang, and H. van Vliet, "Software architecture documentation: The road ahead," in *WICSA*, 2011, pp. 252–255.

[16] H. P. Breivold, I. Crnkovic, and M. Larsson, "Software architecture evolution through evolvability analysis," *Journal of Systems and Software*, Vol. 85, No. 11, 2012, pp. 2574–2592.

[17] J. Derrick and H. Wehrheim, "Model transformations across views," *Sci. Comput. Program.*, Vol. 75, No. 3, 2010, pp. 192–210.

[18] T. Kühne, B. Selic, M.-P. Gervais, and F. Terrier, Eds., *Modelling Foundations and Applications, 6th European Conference, ECMFA 2010, Paris, France, June 15-18, 2010. Proceedings*, ser. Lecture Notes in Computer Science, Vol. 6138. Springer, 2010.

[19] RAVENFLOW, *RAVEN: Requirements Authoring and Validation Environment.* www.ravenflow.com, 2007. [Online]. http://www.ravenflow.com

[20] J. Whitehead, "Collaboration in software engineering: A roadmap," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 214–225. [Online]. http://dx.doi.org/10.1109/FOSE.2007.4

[21] P. Kruchten, P. Lago, H. van Vliet, and T. Wolf, "Building up and exploiting architectural knowledge," in *WICSA*, IEEE Computer Society Washington, DC, USA. IEEE Computer Society, 2005, pp. 291–292.

[22] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. Ali Babar, "A comparative study of architecture knowledge management tools," *Journal of Systems and Software*, Vol. 83, No. 3, 2010, pp. 352–370.

[23] D. Garlan, V. Dwivedi, I. Ruchkin, and B. R. Schmerl, "Foundations and tools for end-user architecting," in *Monterey Workshop*, ser. Lecture Notes in Computer Science, R. Calinescu and D. Garlan, Eds., Vol. 7539. Springer, 2012, pp. 157–182.

[24] I. Gorton, C. Sivaramakrishnan, G. Black, S. White, S. Purohit, C. Lansing, M. Madison, K. Schuchardt, and Y. Liu, "Velo: A knowledge-management framework for modeling and simulation," *Computing in Science Engineering*, Vol. 14, No. 2, march-april 2012, pp. 12 –23.

[25] N. Brown, R. L. Nord, I. Ozkaya, and M. Pais, "Analysis and management of architectural dependencies in iterative release planning," in *WICSA*, 2011, pp. 103–112.

[26] R. L. Nord, I. Ozkaya, and R. S. Sangwan, "Making architecture visible to improve flow management in lean software development," *IEEE Software*, Vol. 29, No. 5, 2012, pp. 33–39.

[27] A. van Hoorn, J. Waller, and W. Hasselbring, "Kieker: a framework for application performance monitoring and dynamic software analysis," in *ICPE*, D. R. Kaeli, J. Rolia, L. K. John, and D. Krishnamurthy, Eds. ACM, 2012, pp. 247–248.

[28] P. Avgeriou, J. Grundy, J. G. Hall, P. Lago, and I. Mistrík, Eds., *Relating Software Requirements and Architectures.* Springer, 2011.

[29] G. Spanoudakis and A. Zisman, "Software traceability: a roadmap," *Handbook of Software Engineering and Knowledge Engineering*, Vol. 3, 2005, pp. 395–428.

[30] A. Egyed and P. Grünbacher, "Automating requirements traceability: Beyond the record & replay paradigm," in *ASE*, IEEE Computer Society Washington, DC, USA. IEEE Computer Society, 2002, pp. 163–171.

[31] P. Kruchten, "Where did all this good architectural knowledge go?" in *ECSA*, ser. Lecture Notes in Computer Science, M. A. Babar and I. Gorton, Eds., Vol. 6285. Springer, 2010, pp. 5–6.

[32] D. Garlan and M. Shaw, "Software architecture:

reflections on an evolving discipline," in *SIGSOFT FSE*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, p. 2.

[33] B. Merkle, "Stop the software architecture erosion," in *SPLASH/OOPSLA Companion*, W. R. Cook, S. Clarke, and M. C. Rinard, Eds. ACM, 2010, pp. 295–297.

[34] A. Chatzigeorgiou, S. Xanthos, and G. Stephanides, "Evaluating object-oriented designs with link analysis," in *ICSE*, A. Finkelstein, J. Estublier, and D. S. Rosenblum, Eds. IEEE Computer Society, 2004, pp. 656–665.

[35] M. Ziane and M. Ó. Cinnéide, "The case for explicit coupling constraints," *CoRR*, Vol. abs/1305.2398, 2013.

[36] R. Koschke, "Software visualization for reverse engineering," in *Software Visualization*, ser. Lecture Notes in Computer Science, S. Diehl, Ed., Vol. 2269. Springer, 2001, pp. 138–150.

[37] J. I. Maletic, A. Marcus, and L. Feng, "Source viewer 3d (sv3d) - a framework for software visualization," in *ICSE*, L. A. Clarke, L. Dillon, and W. F. Tichy, Eds. IEEE Computer Society, 2003, pp. 812–813.

[38] C. S. Collberg, S. G. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A system for graph-based visualization of the evolution of software," in *SOFTVIS*, S. Diehl, J. T. Stasko, and S. N. Spencer, Eds. ACM, 2003, pp. 77–86, 212–213.

[39] S. P. Reiss, "Dynamic detection and visualization of software phases," *ACM SIGSOFT Software Engineering Notes*, Vol. 30, No. 4, 2005, pp. 1–6.

[40] M. D'Ambros, M. Lanza, and M. Lungu, "The evolution radar: visualizing integrated logical coupling information," in *MSR*, S. Diehl, H. Gall, and A. E. Hassan, Eds. ACM, 2006, pp. 26–32.

[41] M. Ogawa and K.-L. Ma, "code_swarm: A design study in organic software visualization," *IEEE Trans. Vis. Comput. Graph.*, Vol. 15, No. 6, 2009, pp. 1097–1104.

[42] K.-L. Ma, "Stargate: A unified, interactive visualization of software projects," in *PacificVis*, IEEE Computer Society Washington, DC, USA. IEEE, 2008, pp. 191–198.

[43] F. Abreu and R. Carapuça, "Object-oriented software engineering: Measuring and controlling the development process," in *Proceedings of the 4th International Conference on Software Quality*, 1994.

[44] J. M. Roche, "Software metrics and measurement principles," *SIGSOFT Softw. Eng. Notes*, Vol. 19, January 1994, pp. 77–85. [Online]. http://doi.acm.org/10.1145/181610.181625

[45] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, Vol. 20, June 1994, pp. 476–493. [Online]. http://portal.acm.org/citation.cfm?id=630808.631131

[46] V. Markovets, R. Dąbrowski, G. Timoszuk, and K. Stencel, "Know thy source code: Is it mostly dead or alive?" in *BCI (Local)*, ser. CEUR Workshop Proceedings, C. K. Georgiadis, P. Kefalas, and D. Stamatis, Eds., Vol. 1036. CEUR-WS.org, 2013, pp. 128–131.

[47] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks*, Vol. 30, No. 1-7, 1998, pp. 107–117.

[48] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, Vol. 2, No. 4, 1976, pp. 308–320.